Computer Science 210
Computer Systems
2006 Semester 2
Lecture Notes

The Alpha Computer Architecture

Bruce Hutton
Department of Computer Science
University of Auckland

Tuesday, January 30, 2007

Contents

1. Overview of Computer Architecture

§1.1 Components and connections

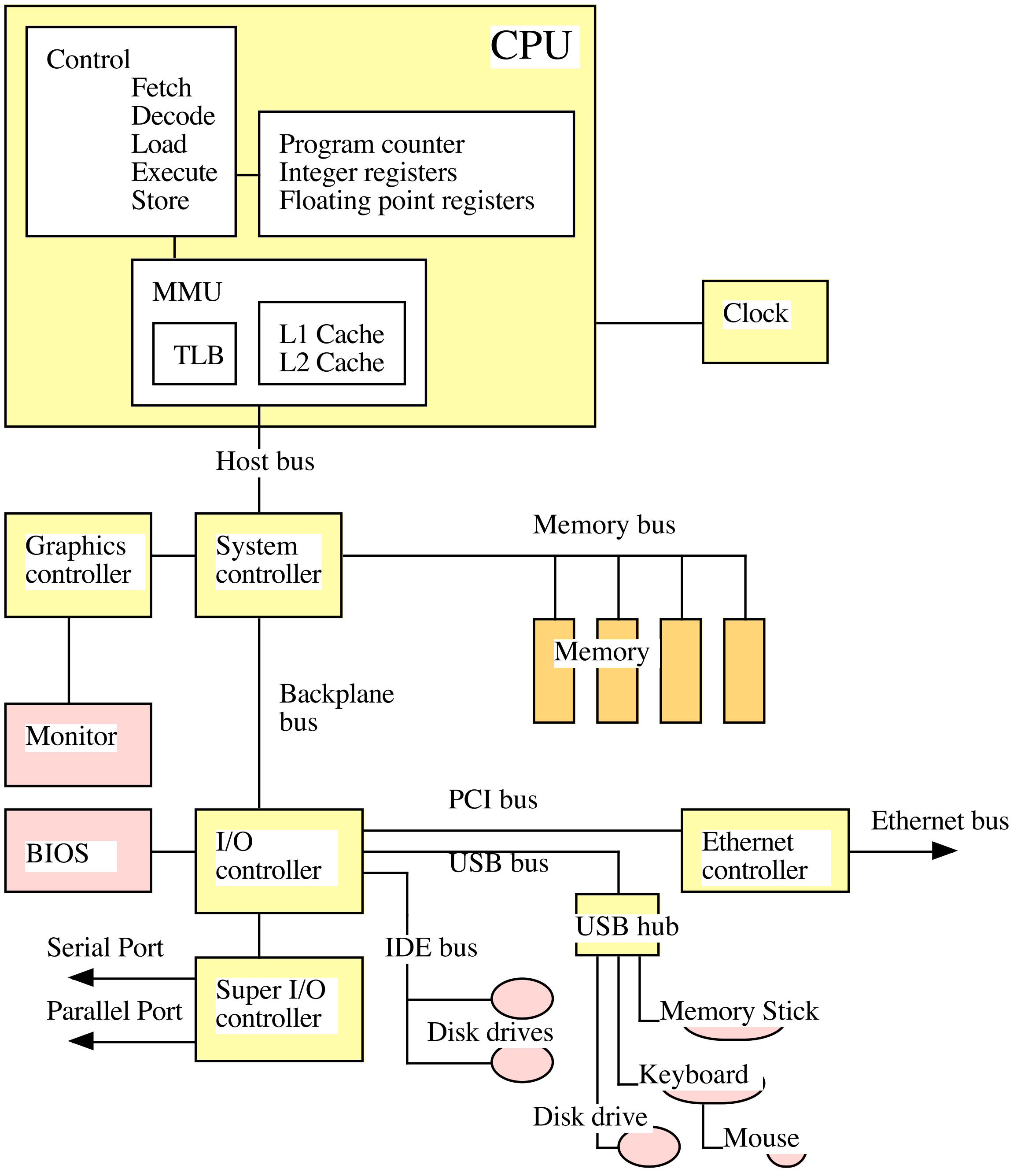
For a very simple introduction to how computers and electronics work, refer to:

http://electronics.howstuffworks.com

http://wikipedia.org

and look for topics involving computers and electronics.

A simple view of a computer would be that it is composed of a CPU, memory, and input/output devices, connected together by buses (sets of wires, used for communication between the components).



§1.2 Creating semiconductor chips

The CPU, memory, and controller chips are composed of packaged **silicon wafers**. The silicon wafers are created by growing a silicon crystal from molten silicon, and slicing it into thin disks. Hundreds of millions of transistors and connecting data and power paths are constructed on the wafer.

Gases can be **infused** into the surface of the chip, by surrounding the chip by an appropriate gas that is absorbed, to create a thin layer with a different chemical composition. For example infusing oxygen into silicon generates SiO_2, which is used as an insulator, and a barrier to ion implantation.

Firing high speed ions at the surface (**ion implantation**) can also be used to insert chemicals into the surface. "**Doping**" elements (P, As, B) are often inserted in this manner.

Chemicals can be **deposited** on the surface, by condensing a hot gas containing the chemical. For example, aluminium or copper can be deposited and later etched to form data and power paths. Si_3N_4, can be deposited to protect the underlying material from ion implantation.

Selective layers of chemicals can be removed (**etched**) by a chemical process. this is often used after photolithography has been used to cover some of the material, and inhibit etching in those areas.

Photosensitive material (**photoresist**) can be deposited on top of a layer of material that we want to etch, and exposed to ultra-violet light or X rays shone through a **mask**, to create a pattern (**photolithography**). The photoresist exposed to the light changes its chemical properties, and either the exposed or unexposed photoresist is then **developed** and removed. The remaining photoresist is **hardened**, and a chemical process is used to remove the exposed underlayer not covered by the photoresist. The material exposed by removing the underlayer can then have a doping material implanted. Finally, the remaining hardened photoresist and underlayer beneath it can be removed.

The wafer can be **polished**, to remove material that projects above the surface, and create a flat plane (**planarization**). For example, holes can be etched, then the surface covered with another material, then the surface polished to remove the material other than that filling the holes.

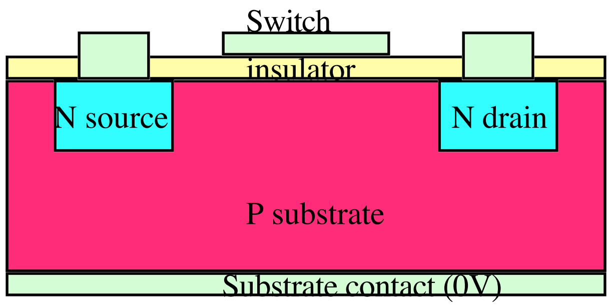
By performing the above processes many times, different materials can be added to different parts of the wafer, and transistors built up. Perhaps about 450 processes, including about 30 masks may be performed and 20 different layers may be built up.

In 2006, transisters have dimensions of about 65 nanoMetres, which is only a few hundred atoms across. In 2007, Intel hope to produce 45 nanoMetre transisters. There is talk of decreasing the dimensions down to around 20 nanoMetres in the future, which must be getting close to fundamental limits on size. Something else, such as building more layers, or making thinner connectors, will have to be done to pack more transistors on a chip.

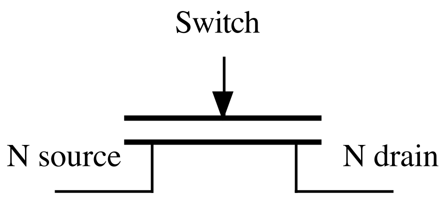
Silicon has 4 valence electrons. **Phosphorus** or **arsenic**, with **one more valence electron**, can be used for **N-type (negative) doping**. **Boron** or **gallium**, with **one less valence electron**, can be used for **P-type (positive) doping**. N-type doping generates a material with free electrons that can easily move out of the material. P-type doping generates a material with "holes" for electrons, that can easily attract external electrons.

I	II		III	IV	V	VI	VII	VIII
H								He
Li	Be		**B**	C	N	O	F	Ne
Na	Mg		Al	**Si**	**P**	S	Cl	Ar
K	Ca	...	**Ga**	Ge	**As**	Se	Br	Kr

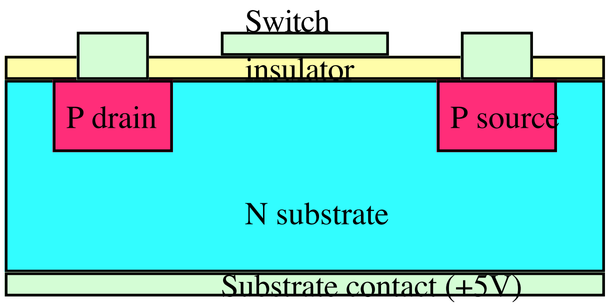
When N-type and P-type material are put together, they create a **diode**, which is essentially a one way gate. If the N-type material is connected to a low voltage, and the P-type material is connected to a high voltage, the electrons will flow through the connection, but not if the voltage difference is the other way around. It is possible to create more complicated transistors (metal-oxide semiconductor field-effect transistors, MOSFET), that allow electricity to flow or not flow, depending on the voltage supplied to a "switch". There are two types of such transistor, P-type (switch on with low voltage) and N-type (switch on with high voltage).



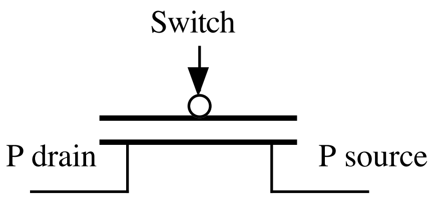
N-type FET Transistor



Symbolic Representation

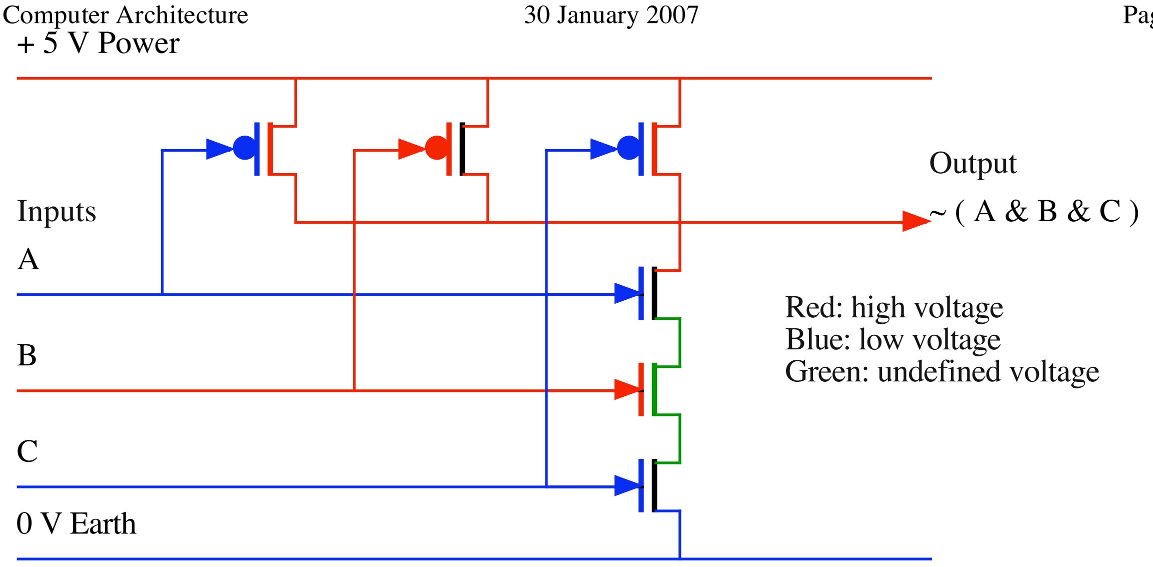


P-type FET Transistor

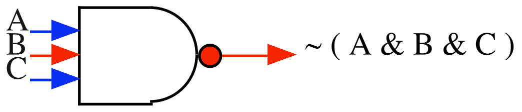


Symbolic Representation

By using such circuits, we can create higher level NAND, NOR, and NOT gates:



Circuit representing a CMOS NAND gate, with 3 inputs



Symbolic representation of NAND gate with three inputs

§1.3 Creating Boolean functions out of gates

In fact any Boolean function can be built from transistors. For example, an integer value is essentially composed of Boolean bit values. The bits that make up the sum of two integer values are functions of the bits that make up the integers being added, so a circuit to add two integer values can be built from transistors.

A “half adder” is a logic circuit that takes two binary digits, and computes their sum (the “exclusive or” of the bits), and the carry (the “and” of the bits). For example, 1 + 0 = 1, with carry 0, and 1 + 1 = 0, with carry 1.

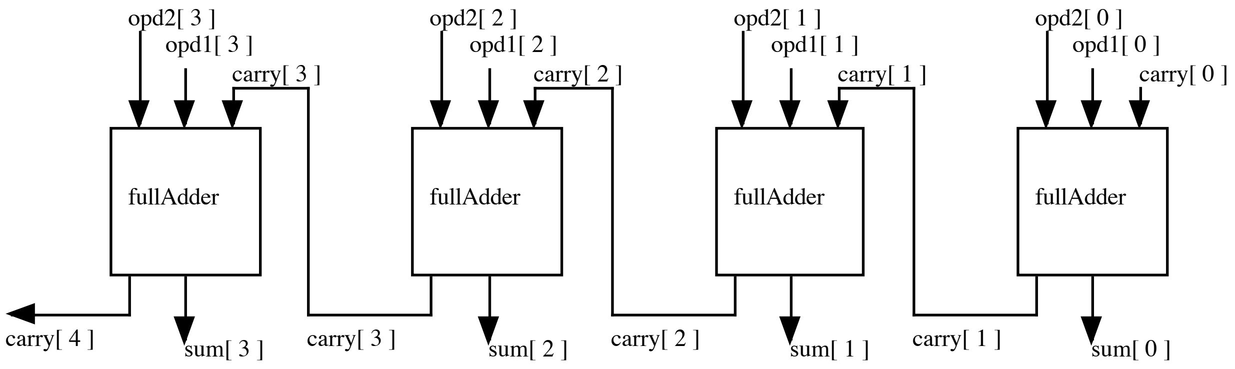
```
component { in opd1, opd2 } halfAdder { out sum, carry }
     begin
          { in opd1 opd2 } xor( 2 ) { out sum };
          { in opd1 opd2 } and( 2 ) { out carry };
     end
```

Two half adders can be combined to produce a “full adder”, that takes two binary digits, together with a carry in, and generates the sum and carry out. For example, if we have a carry in of 1, and add 1 + 1, we get 1, with a carry out of 1.

```
component { in opd1, opd2, carryIn } fullAdder { out sum, carryOut }
     begin
          path sum1, carry1, carry2;
          { in opd1, opd2 } halfAdder { out sum1, carry1 };
          { in sum1, carryIn } halfAdder { out sum, carry2 };
          { in carry1 carry2 } or( 2 ) { out carryOut };
     end
```

By combining an array of “n” full adders, we can add two “n” bit numbers. The carry out from adding the “i”th bits becomes the carry in when adding the “i+1”th bits, so the carry ripples through the circuit, and the component is called a “ripple adder”. The algorithm executes in O(n) time.

```
component { in opd1[ n ], opd2[ n ], carryIn } add( n )
     { out sum[ n ], carryOut }
     begin
          path carry[ n + 1 ];
          { in carryIn } join( 1 ) { out carry[ 0 ] };
          for i from 0 upto n do
               { in opd1[ i ], opd2[ i ], carry[ i ] } fullAdder
                    { out sum[ i ], carry[ i + 1 ] };
          end
          { in carry[ n ] } join( 1 ) { out carryOut };
     end
```



§1.4 Flip-Flops

We can create what is called a “flip-flop” to store a “bit” (binary digit). This is a logic circuit that has feedback (cycles in the directed graph of components and paths) that provides an internal state. An array of flip-flops can be used to represent the value of a register.

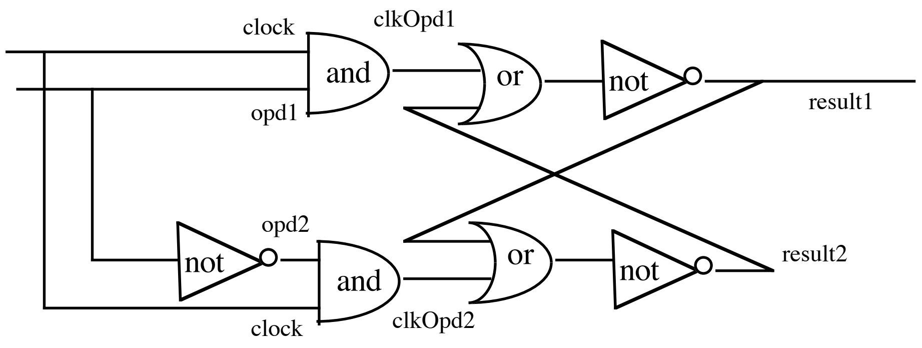
A simple flip-flop takes a clock signal “clock”, and a value “opd1” as inputs, and produces a value “result1” as output.

If “clock == true”, and “opd1 == true”, then “result2 = false”, and “result1 = true”. If “clock == true”, and “opd1 == false”, then “result1 = false”, and “result2 = true”. So if “clock == true”, “result1 = opd1”, and “result2 = !opd1”.

If “clock = false”, then “result1” and “result2” can take any value, so long as “result2 == !result1”.

So, when the clock is set, a simple flip-flop stores the value of “opd1” in “result1”. The value remains there, even after the clock is cleared, and “opd1” changes.

```
component { in clock, opd1 } simpleFlipFlop { out result1 }
     begin
          path opd2, clkOpd1, clkOpd2, result2;
          { in opd1 } not( 1 ) { out opd2 };
          { in clock opd1 } and( 2 ) { out clkOpd1 };
          { in clock opd2 } and( 2 ) { out clkOpd2 };
          { in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };
          { in clkOpd2 result2 } or( 2 ).not( 1 ) { out result1 };
     end
```



To allow the input data to stabilise before the change is visible to the output, and to avoid problems when the output feeds back to the input, it is best to pair two simple flip-flops, to form a "master-slave flip-flop". When "clock1" is true, the value of "opd" is transferred to the internal state "value", but does not pass through to the output "result". When "clock1" is false, the internal state "value" is transferred to "result", but changes in the input have no effect. Thus the master-slave flip-flop appears to transfer the data from "opd" to "result" when "clock1" changes from true to false.

```
component { in clock1, opd } masterSlaveFlipFlop { out result }
     begin
          path clock2, value;
          { in clock1 } not( 1 ) { out clock2 };
          { in clock1, opd } simpleFlipFlop { out value };
          { in clock2, value } simpleFlipFlop { out result };
     end
```

§1.5 Registers and memory

The CPU (central processing unit) contains electrical circuits, to decode and execute instructions, load data from and store data to memory, etc. It also contains a small amount of fast local memory, namely what are called **registers**. On the Alpha, all registers are composed of 64 bits. There is a **program counter (PC) register**, that contains the memory address of the next instruction, **32 integer registers** to contain integer data, and **32 floating point register**s, to contain floating point data. There are also other internal registers, to store temporary information, etc.

Thus we can think of our CPU as roughly corresponding to the data structure

```
class CPU {
     Quadword programCounter;
     Quadword[] intReg = new Quadword[ 32 ];
     Quadword[] floatReg = new Quadword[ 32 ];
     // and some special registers
     }
```

In fact `intReg[ 31 ]` and `floatReg[ 31 ]` always return zero if read, and writing to them has no effect.

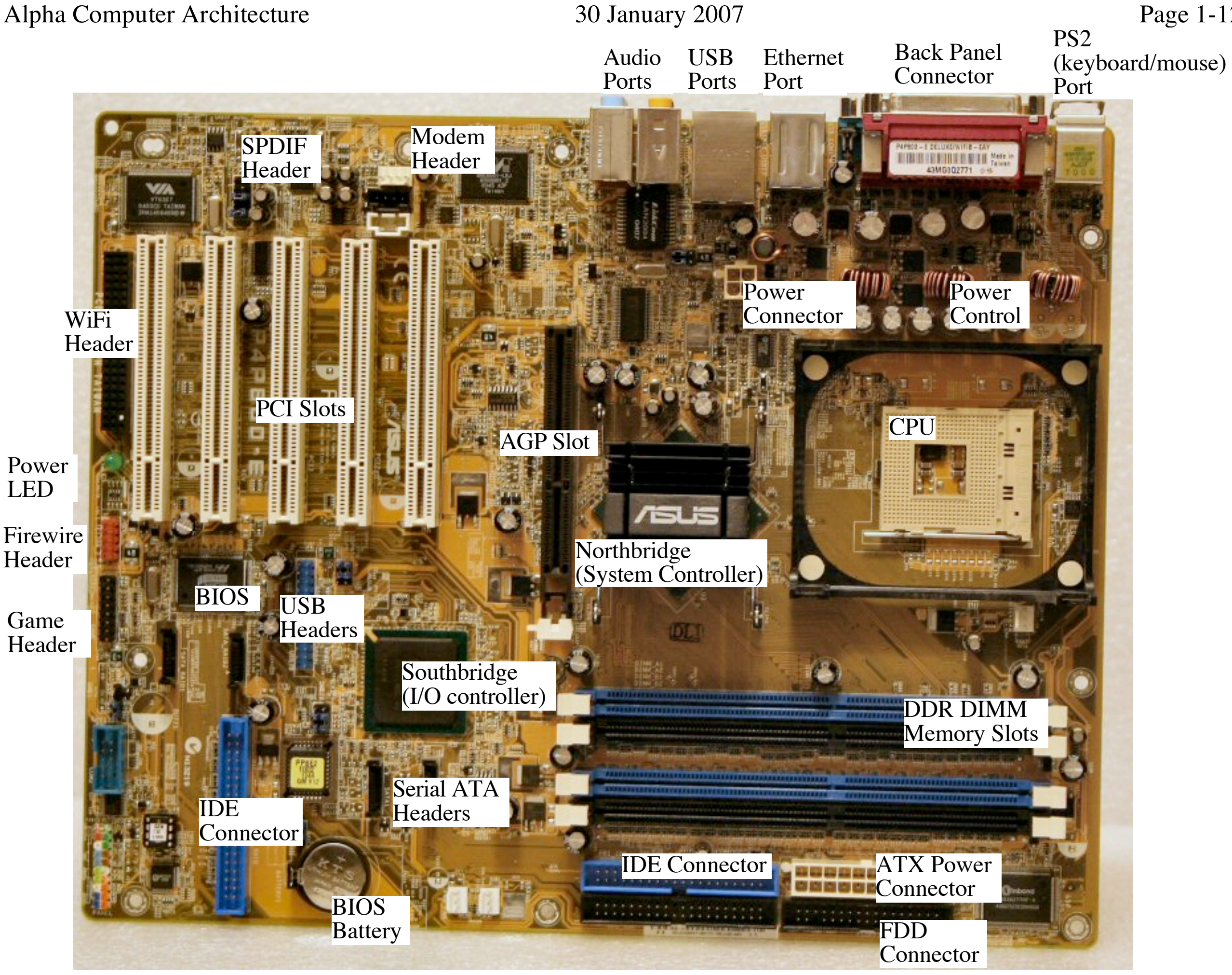
Memory corresponds to an array of bytes

```
     byte[] memory = new byte[ ... ];
```

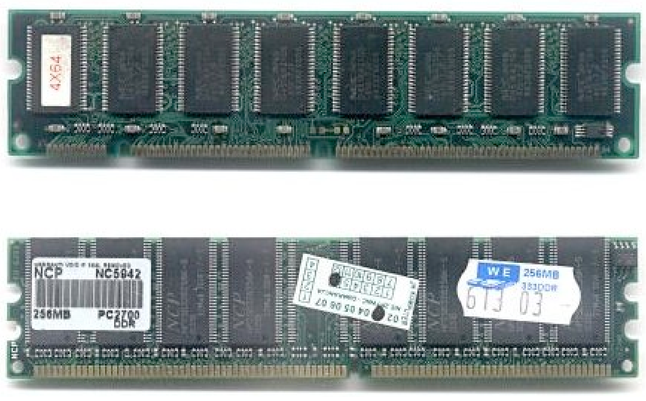
that can be indexed by a memory address.

§1.6 Motherboards, cards and buses

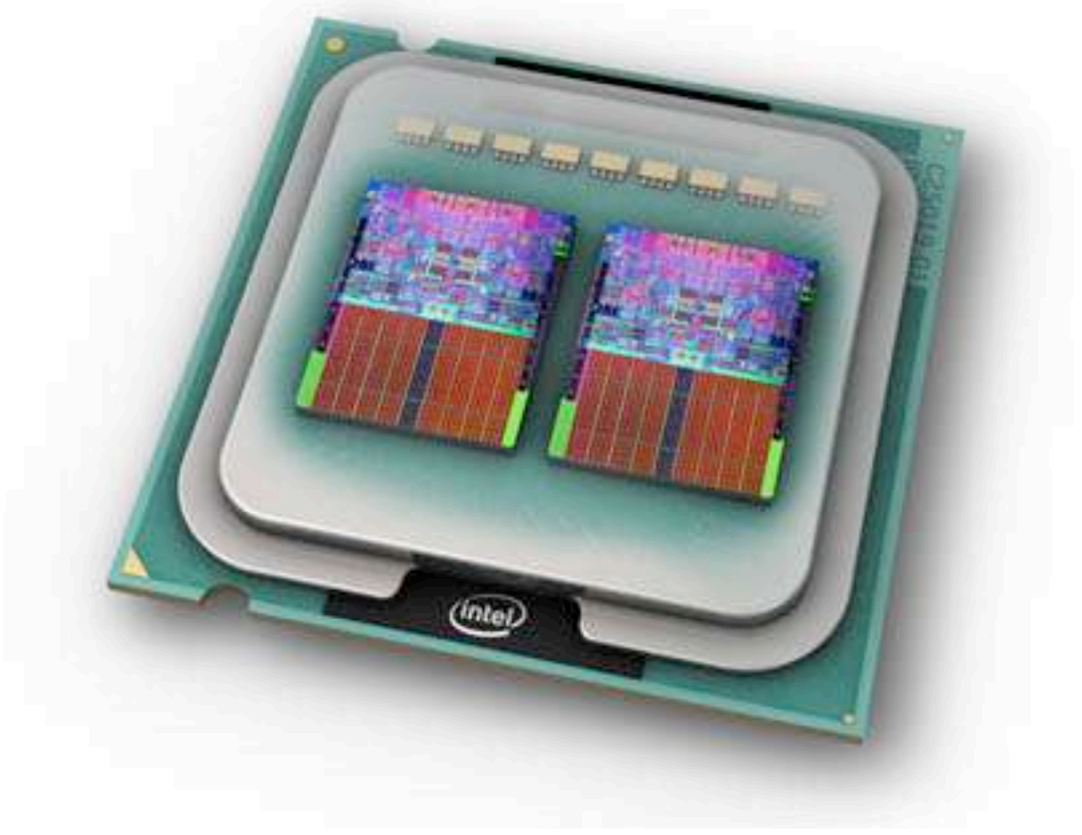
Inside your computer, you will find a large green circuit board, called the motherboard. The motherboard connects all the components of the computer together. Smaller boards called cards can be plugged into the motherboard. Computer chips, including the CPU, memory, and input/output controllers are plugged in to the boards. The CPU is usually surrounded by large cooling fins with a fan. There is usually a central hub, that all communications between components pass through. The wires that connect the components are called buses.



A motherboard



Memory cards

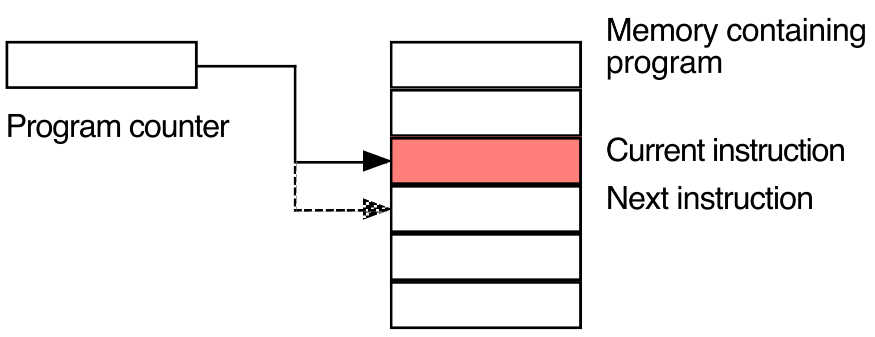


A drawing representing a processor chip containing two dies (silicon wafers). The regular area at the bottom of each die is the cache memory.

§1.7 The execution cycle

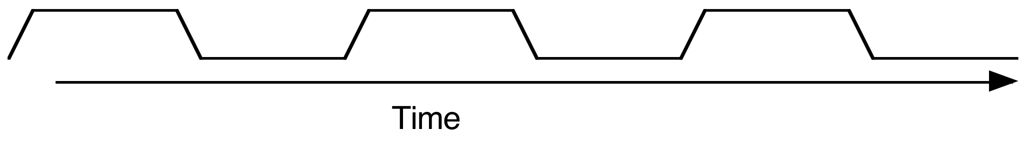
Instructions are stored in computer memory. The program counter register contains the memory address of the next instruction. The CPU loops, obtaining an instruction, incrementing the program counter, decoding and executing the instruction, etc.

```
while ( true ) {
     Longword instruction = getLongwordAt( programCounter );
     programCounter = programCounter + 4;
     decode the instruction;
     obtain the operands of the instruction;
     perform the operation of the instruction;
     save the result;
     }
```



§1.8 The clock

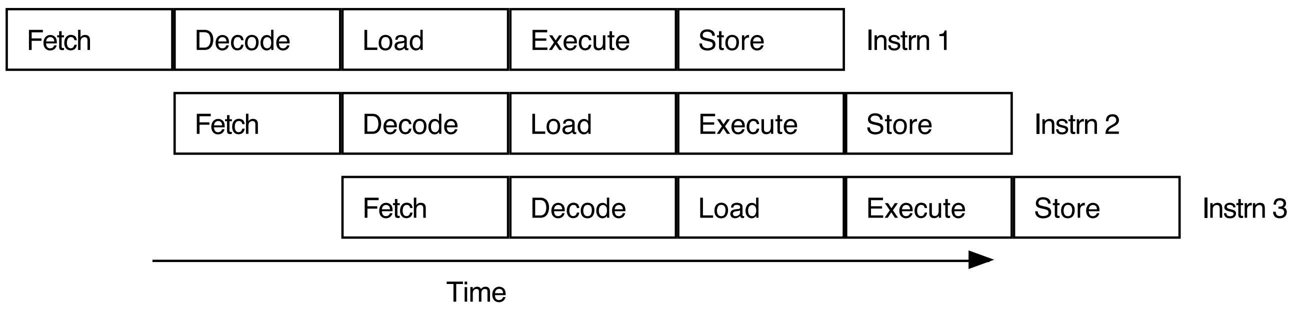
There is a processor clock that generates a step function with regular changes in voltage. Perhaps the change from low to high voltage represents a clock tick. This clock is used to control the activity of the CPU. A typical CPU clock speed in 2006 is about 3.8GHz, or 0.26 nanoSec for a clock cycle.



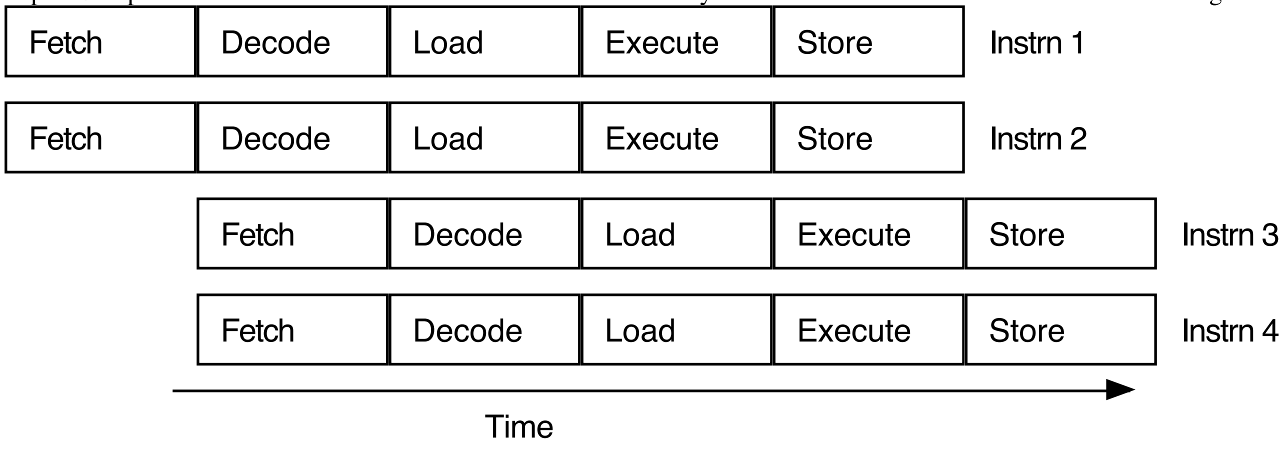
All activity in the CPU is triggered by a clock tick, and data paths are opened and closed to permit data to flow through from one part of the CPU to another. Excluding memory accesses, simple instructions might take about 15-20 clock cycles to execute (say 5-6 clock cycles for an instruction fetch, 3-4 cycles to obtain the operands, 2-3 cycles to compute the the result, and 3-6 cycles to save the result).

§1.9 Overlapping instruction execution

Nowadays, instruction execution is "**pipelined**", and execution of instructions overlap, so that when one instruction is being executed, the next instruction is being decoded, and the one after that is being fetched. In fact, if there are multiple copies of the circuitry in the CPU, several instructions may even be "**issued**" (scheduled to execute) **at the same time**. Combined with pipelining, perhaps a total of up to 50 instructions may be in the process of execution at the same time. However, conditional branch instructions may limit the extent to which instruction execution may be pipelined, because it is not possible to determine which instruction will be executed next, until after the previous instruction has completed execution (although we can guess which one will be executed, execute our guess, and discard computations if the guess turns out to be wrong). Similarly, the operands for one instruction may depend on the results of previous instructions, and hence an instruction might have to wait for the result of a previous instruction. Also, instructions can only execute in parallel if independent circuits are available for use.



Pipelining of instructions



Dual issue (concurrent scheduling of instructions)

§1.10 Cache

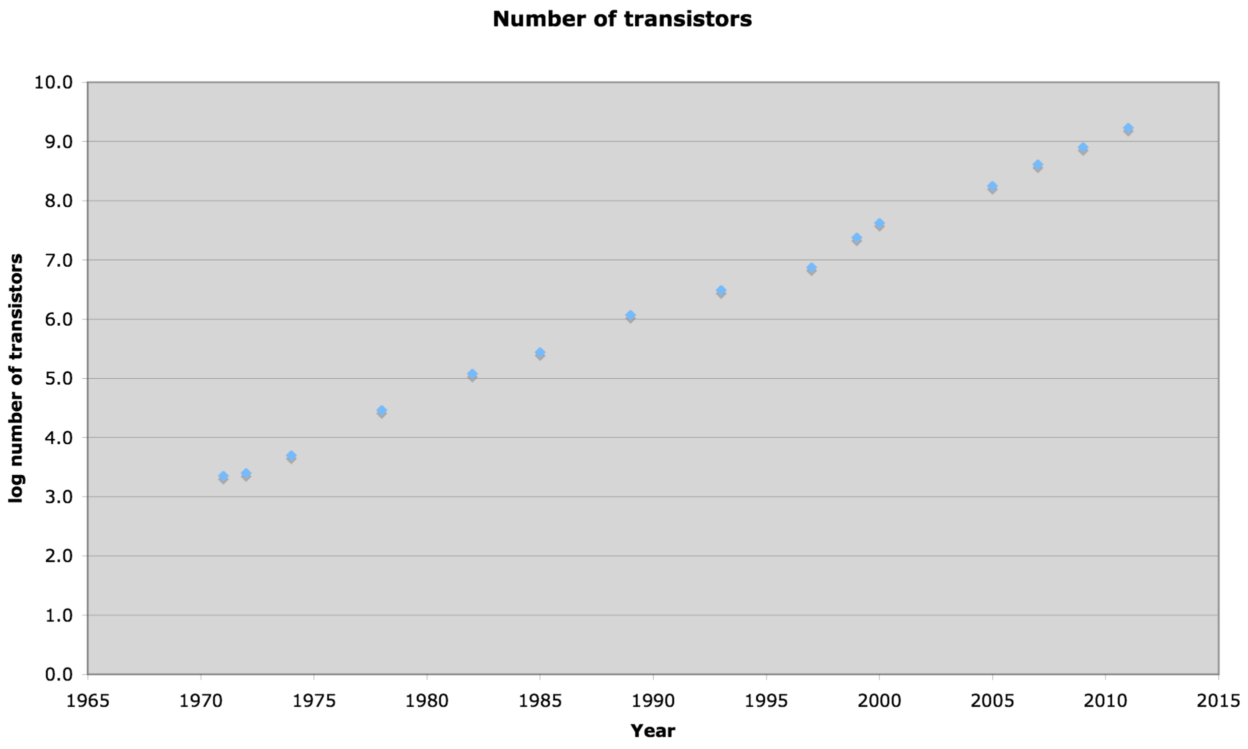
External memory, stored on separate chips, takes much longer for the CPU to access than registers. External memory has its own clock, which in 2006 is about 667 MHz, much slower than the CPU clock. To decrease the delay in external memory access, the CPU "**caches**" (keeps a copy of) recently used memory. The memory in the CPU used for the cache (**static RAM**) is faster, but requires more transistors and is more expensive to build than the external memory (**dynamic RAM**). Nowadays, there are at least two levels of cache. In 2006, the smallest and fastest (level 1) cache is about 56 KBytes in size, and takes about 2 CPU clock cycles to access. Level 2 cache is is about 512 KBytes - 2 MByte in size, and takes about 6-10 clock cycles to access, if part of the CPU. Some CPUs even have a level 3 cache (around 8MBytes). External memory is much larger, about 512 MBytes - 1 GByte, and takes about 100 - 300 CPU clock cycles to access. But access to memory is fast compared with disk access times. The seek time (time to move the disk head to the right track) is around 5 - 10 milliSec, and the rotation delay, while the track spins to the correct sector of the track is similar. This is tens of millions of clock cycles! But hard disks provide large amounts of permanent storage – 160 - 250 GBytes is fairly typical for a personal computer in 2006, and Weta Workshop, the special effects company for "Lord of the Rings", have hundreds of teraBytes of disk space.

Registers	32 - 64	1 cycle
L1 cache	56 KB	2 cycles
L2 cache	512 KB - 2MB	6 - 10 cycles
External Memory	512MB - 1 GB	100 - 300 cycles
Disks	160 GB - 250 GB	10^7 cycles to seek

§1.11 Moore's law

As a general rule, the number of transistors that can fit on a processor chip doubles every couple of years, due to the increased ability to manufacture smaller features on a chip. The number of transistors on a memory chip doubles about every 18 months.

Intel Processor	Year of introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium IV processor	2000	42,000,000
Pentium IV processor	2005	178,000,000
	2007?	410,000,000



Year	Feature size (nm)
1980	3000
1983	2000
1984	1500
1987	1000
1989	600
1991	500
1992	400
1994	350
1997	250
1998	180
2002	130
2004	90
2006	65
2007	45

Because everything is smaller, it takes less time for information to propagate between components. Because more transistors are available, the circuitry on the chip can be duplicated, and instructions can be executed in parallel. More space is available on the chip to implement more complex algorithms. More space is available for such things as cache memory (which takes up about a third of the space in a modern CPU). So the performance of computers also increases at a comparable rate (perhaps even faster). The density of storage of data on a hard disk also increases at a similar rate. However, disk access times depend primarily on the time taken to position the disk head, so disk “latency” (the time delay before the data can be accessed) does not change much.

CPU manufacturers are now finding it more difficult to achieve speed improvements by increasing the parallelism withing a single processor, or increasing the size of cache, and are now tending to develop chips with multiple processors instead.

§1.12 Speed of execution

It is important to have an appreciation of how much difference there is in the times taken to perform various operations in a computer. Disk movements occur at speeds comparable to the 1/10th of the speed of sound, while electronic signals within the CPU travel at speeds around 2/3 of the speed of light.

10^0 (1 second)	Time for light to travel to the moon.
10^{-1}	Blink of an eye. Duration for a frame of a movie. Sattelite communication delay.
10^{-2}	Disk seek time. Time for a disk to rotate. TV refresh rate.
10^{-3} (1 millisecond)	Time for sound to travel 30cm. Sound frequency.
10^{-4}	Time to transfer 1 byte over 56Kb/sec modem.
10^{-5}	
10^{-6} (1 microsecond)	Time to transfer 1 byte over broadband modem.
10^{-7}	Time to transfer 1 byte over ethernet. Time to access memory. Time to transfer 1 byte from disk.
10^{-8}	
10^{-9} (1 nanosecond)	CPU clock speed. Time to execute an

	instruction. Time for light to travel 30cm.
10^{-10}	Microwave frequency

§1.13 Size of computer data

It is also nice to get an idea of how small the components of a computer are.

10^{0} (1 metre)	Wavelength of VHF/UHF.
10^{-1}	Wavelength of audible sound.
10^{-2} (1 centimetre)	Wavelength of microwaves.
10^{-3} (1 millimetre)	
10^{-4} (100 micrometre)	Width of human hair. Dimensions of a dust mite.
10^{-5} (10 micrometre)	Dimensions of a eukaryotic cell.
10^{-6} (1 micrometre)	Dimensions of a bacterium. Wavelength of visible light.
10^{-7} (100 nanometre)	Dimensions of a bit in a computer memory or on disk. Dimensions of a transistor. Dimensions of a virus.
10^{-8} (10 nanometre)	
10^{-9} (1 nanometre)	Wavelength of soft X-rays.
10^{-10} (1 Angstrom)	Dimensions of an atom.

2. Alpha Instruction Formats

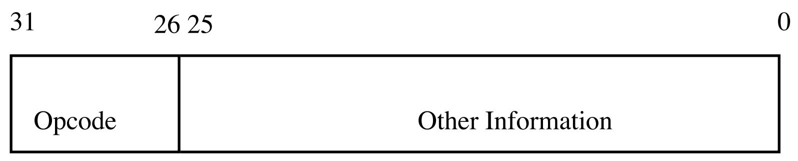
On the Alpha, each instruction is stored in a longword, and hence is composed of 32 bits or 4 bytes. Instructions must be aligned (stored at an address divisible by 4).

Normally, instructions at successive addresses are executed in sequence (or at least appear to be), because the program counter is incremented by the size of an instruction after loading the instruction into the CPU. However, some instructions (called branch instructions) can modify the program counter. This is how we deviate from a straight line path of execution and manage to create loops, if statements, etc.

Many modern machines have the following kinds of instructions:

- **Instructions that perform arithmetic and logical operations on registers.** For example, there might be an instruction to add the contents of two registers, and store the result in a third register. On the Alpha, we could write "`addq $0, $1, $2;`" to generate an instruction that adds the quadwords in integer registers 0 and 1, and stores the result in register 2.
- **Instructions that load data from memory into a register, or store data from a register into memory.** For example, there might be an instruction to load a quadword from memory into an integer register, or save the contents of an integer register into a quadword in memory. On the Alpha we could write "`ldq $2, 0($1);`" to generate an instruction that loads the quadword at the address specified by the contents of integer register 1, into integer register 2.
- **Instructions that check the value of a register, and, based on this value, either do nothing, or modify the program counter, so that the next instruction is obtained from a different place.** On the Alpha we could write "`bne $1, loop;`" to generate an instruction that checks the value of integer register 1, and if it is not equal to 0, changes the program counter to the address corresponding to the label "`loop`".

On the Alpha, all instructions have a 6 bit opcode stored in bits 26-31, which indicates the kind of instruction. Given this opcode, the CPU knows how to decode the rest of the instruction.

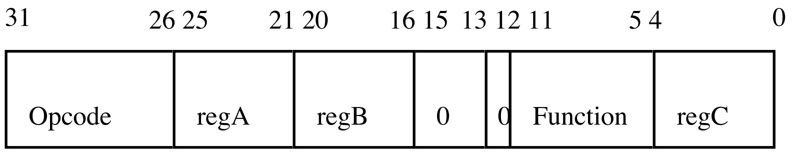


Common format for all instructions

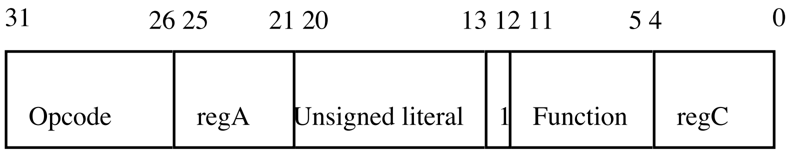
Opcodes

00	call_pal	10	inta	20	ldf	30	br
01	call_xfc	11	intl	21	ldg	31	fbeq
02	Res	12	ints	22	lds	32	fblt
03	Res	13	intm	23	ldt	33	fble
04	Res	14	itfp	24	stf	34	bsr
05	Res	15	fltv	25	stg	35	fbne
06	Res	16	flti	26	sts	36	fbge
07	Res	17	fltl	27	stt	37	fbgt
08	lda	18	misc	28	ldl	38	blbc
09	ldah	19	hw_mfpr	29	ldq	39	beq
0a	ldbu	1a	jsr	2a	ldl_l	3a	blt
0b	ldq_u	1b	hw_ld	2b	ldq_l	3b	ble
0c	ldwu	1c	fpti	2c	stl	3c	blbs
0d	stw	1d	hw_mtpr	2d	stq	3d	bne
0e	stb	1e	hw_rei	2e	stl_c	3e	bge
0f	stq_u	1f	hw_st	2f	stq_c	3f	bgt

The integer operate instruction formats on the Alpha are shown below.



Integer operate instruction with second operand a register



Integer operate instruction with second operand a literal

Integer operate instructions also have a function code, stored in bits 5-11, which gives more detail about what operation the instruction should perform. The operands are specified in other fields. Bits 21-25 specify the register that contains the first operand. The second operand can be either a register or a constant, and the appropriate alternative is specified by a flag in bit 12. If the flag is 0, the second operand is a register, and this register is specified in bits 16-20. If the flag is 1, the second operand is an unsigned 8 bit constant, and this constant is specified in bits 13-20. The result is stored in a register, and the destination register is specified in bits 0-4. There are similar formats for other instructions.

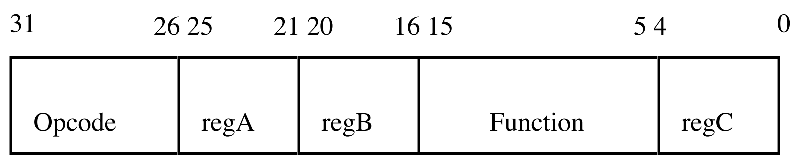
The size of each field in the instruction is important. Because the field for a register number contains 5 bits, we can specify at most 2^5 = 32 different registers. Thus it is not possible to have more than 32 integer registers. The field for the unsigned constant is 8 bits, so must be in the range 0 .. 255.

It must be possible to determine the meaning of an instruction by a straightforward algorithm. The opcode determines the overall format of the rest of the instruction. Given that we know the instruction is an integer operate instruction, we can check the function code to determine exactly which integer operate instruction it is, and check the literal flag to determine whether the second operand is a constant or register.

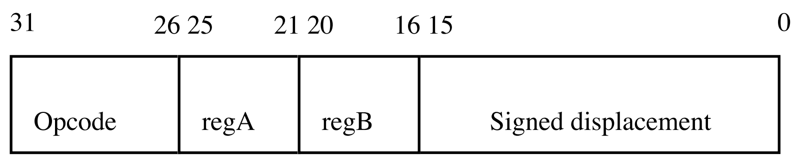
Function code for opcode 0x10

00	addl	20	addq	40	addlv	60	addqv
01		21		41		61	
02	s4addl	22	s4addq	42		62	
03		23		43		63	
04		24		44		64	
05		25		45		65	
06		26		46		66	
07		27		47		67	
08		28		48		68	
09	subl	29	subq	49	sublv	69	subqv
0a		2a		4a		6a	
0b	s4subl	2b	s4subq	4b		6b	
0c		2c		4c		6c	
0d		2d	cmpeq	4d	cmplt	6d	cmple
0e		2e		4e		6e	
0f	cmpbge	2f		4f		6f	
10		30		50		70	
11		31		51		71	
12	s8addl	32	s8addq	52		72	
13		33		53		73	
14		34		54		74	
15		35		55		75	
16		36		56		76	
17		37		57		77	
18		38		58		78	
19		39		59		79	
1a		3a		5a		7a	
1b	s8subl	3b	s8subq	5b		7b	
1c		3c		5c		7c	
1d	cmpult	3d	cmpule	5d		7d	
1e		3e		5e		7e	
1f		3f		5f		7f	

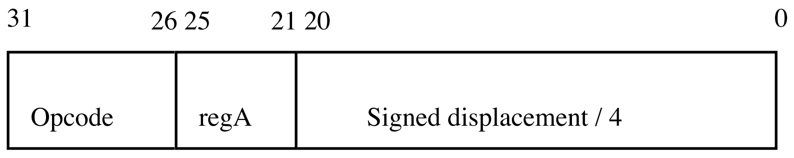
The other instruction formats on the Alpha are shown below.



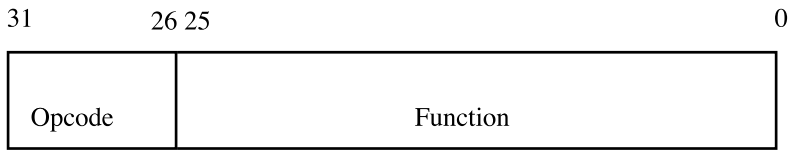
Floating point operate instruction



Memory access instruction



Branch instruction



Special instruction

There are some instructions that ignore some fields, or use them for some other purpose, but all instructions still more or less conform to one of these formats.

3. Assembly language

How do we create machine code? We write a program, in textual form, and use another program to translate this into machine code instructions. The name given to the translator depends on how high level the textual version of the program is. If the original program is written in a typical high level language, such as C, and is fairly machine independent, then the translator is called a compiler. If the original program is written in a low level language that does little more than specify the instructions in a textual form, the language is called assembly language, and the translator is called an assembler. Because different kinds of computers have different instruction sets, assembly language is very different on different kinds of computers.

What does an assembly language program look like? Here is a very simple piece of assembly language:

```
entry main.enter;

import "../IMPORT/callsys.h";

//   void main() {
//       while ( TRUE ) {
//           char c;
//           c = getChar();
//           if ( c < 0 )
//               break;
//           putchar( c );
//           }
//       exit( 0 );
//       }
block main uses CALLSYS {
     code {
     public enter:
               {
          loop:
               ldiq $a0, CALLSYS_GETCHAR;
               call_pal  CALL_PAL_CALLSYS;
               blt       $v0, end;
               mov       $v0, $a1;
               ldiq $a0, CALLSYS_PUTCHAR;
               call_pal  CALL_PAL_CALLSYS;
               br        loop;
          end:
               }
               {
               clr       $a1;
               ldiq $a0, CALLSYS_EXIT;
               call_pal  CALL_PAL_CALLSYS;
               }
          } code
     } block main
```

It should be pointed out that this is my own home grown assembler. It is a bit different from most conventional assemblers.

Altogether, the program reads characters from the keyboard, and outputs them to the screen.

The line

```
entry main.enter;
```

just specifes the entry point for the program (in other words, where the program starts executing).

The line

```
import "../IMPORT/callsys.h";
```

specify that the code in the specified file is imported (included).

The C-like code, with `//` to the left of each line, is really just a sequence of comments. They are there for people, but the assembler ignores them. Because assembly language programs are very low level and difficult to read, it is desirable to always document your assembly language program by comments written in a high level language.

The lines

```
block main uses CALLSYS {
     } block main
```

just specify that we are creating a block of code called main, using some definitions in a block called register that specifies the register numbers for the symbolic names `a0`, `a1`, `v0`, etc, and using some definitions in a block called `CALLSYS` that specifies the values of `CALL_PAL_CALLSYS`, `CALLSYS_GETCHAR`, `CALLSYS_PUTCHAR`, etc.

The lines

```
     code {
          } code
```

just specify that we are defining code (instructions), rather than data. The assembled bit patterns are placed in the section of memory used for code.

The line

```
     public enter:
```

labels some code, with the name "`enter`". This code can be referred to outside the block, as `main.enter`.

The lines

```
          {
     loop:
          ldiq $a0, CALLSYS_GETCHAR;
          call_pal  CALL_PAL_CALLSYS;
          blt       $v0, end;
          mov       $v0, $a1;
          ldiq $a0, CALLSYS_PUTCHAR;
          call_pal  CALL_PAL_CALLSYS;
          br        loop;
     end:
          }
```

represent the real work.

We label the beginning of the loop by the identifier "`loop`". As in all computer languages, the name is arbitrary, and could be consistently replaced by any other identifier.

The line

```
          ldiq      $a0, CALLSYS_GETCHAR;
```

loads the constant value `CALLSYS_GETCHAR` (whatever that has been defined to be in the block `CALLSYS`, in "`../IMPORT/callsys.h`") into register `a0`.

The line

```
          call_pal  CALL_PAL_CALLSYS;
```

then makes a request (rather like a function invocation) to the operating system to do something (read a character from the keyboard). The operating system uses the value in register `$a0` to determine what action to perform (in this case read a character), and returns the result in register `$v0`. Other parameters to system calls may be passed in registers `$a1`, `$a2`, `$a3`, ...

The line

```
          blt       $v0, end;
```

causes a branch out of the loop if the returned characacter is < 0 (indicating end of file).

The lines

```
          mov       $v0, $a1;
          ldiq      $a0, CALLSYS_PUTCHAR;
          call_pal  CALL_PAL_CALLSYS;
```

move the character from register `v0` into register `a1`, load the constant value `CALLSYS_PUTCHAR` into register `$a0` (to specify that the action is to write a character), and makes a request to the operating system (to write the character to the screen).

The line

```
          br        loop;
```

causes the program to branch back to the address corresponding to the label `loop`.

The lines

```
          {
          clr       $a1;
          ldiq $a0, CALLSYS_EXIT;
          call_pal  CALL_PAL_CALLSYS;
          }
```

Cause the program to terminate and control to be returned to the operating system.

Now, to really understand this program, you have to not only understand what each instruction does, but also know the general conventions for making system calls, and what the two system calls that read and write a character, and the exit system call actually do.

There is also the additional complication that many of these instructions are not real instructions, and get converted into something different. However, it still gives a feeling for the manner in which assembly language programs are written, and the low level they correspond to. We have to build loops and if statements out of branch instructions.

In fact the instructions for making system call requests are usually put inside functions, and the functions are called instead.

```
block Sys {
     //   char getChar() {
     //        //   read a character from the simple terminal;
     //        }

     public block getChar uses proc, CALLSYS {
          code {
          public enter:
               lda       $sp, -sav0($sp);
               stq       $ra, savRet($sp);
          body:
               ldiq $a0, CALLSYS_GETCHAR;
               call_pal  CALL_PAL_CALLSYS;
          return:
               ldq       $ra, savRet($sp);
               lda       $sp, +sav0($sp);
               ret;
               } code
          } block getChar

     //   void putChar( char c ) {
     //        //   write a character to the simple terminal;
     //        }
     public block putChar uses proc, CALLSYS {
          code {
          public enter:
               lda       $sp, -sav0($sp);
               stq       $ra, savRet($sp);
          body:
               mov       $a0, $a1;
               ldiq $a0, CALLSYS_PUTCHAR;
               call_pal  CALL_PAL_CALLSYS;
          return:
               ldq       $ra, savRet($sp);
               lda       $sp, +sav0($sp);
               ret;
               } code
          } block putChar
     ...
     } block main
```

The code

```
               lda       $sp, -sav0($sp);
               stq       $ra, savRet($sp);
```

at the beginning and

```
               ldq       $ra, savRet($sp);
               lda       $sp, +sav0($sp);
               ret;
```

at the end is code to save registers on entry to a function and restore them on exit. Don't worry about this code for the moment.

Let's consider another piece of assembly language that uses the `putChar` function.

```
block IO {
     ...
     //   void print( char *s ) {
     //        while ( *s != 0 ) {
     //             putChar( *s );
     //             s++;
     //             }
     //        }
     //
     public block print uses proc {
          abs {
               s         =    s0;
               } abs
          code {
          public enter:
               lda       $sp, -sav1($sp);
               stq       $ra, savRet($sp);
               stq       $s0, sav0($sp);
          body:
               mov       $a0, $s;                 //   Pointer to char in string
               {
               while:
                    ldbu $a0, ($s);               //   Get character
                    beq       $a0, end;           //   Break if at end of string
               do:
                    bsr       Sys.putChar.enter;  //   Print char
                    addq $s,  1;                  //   Increment pointer
                    br        while;
               end:
               }
```

```
        return:
             ldq       $s0, sav0($sp);
             ldq       $ra, savRet($sp);
             lda       $sp, +sav1($sp);
             ret;
             } code
        } block print
   ...
```

Again, don't worry about the function entry/exit code.

This assembly language represents a function, that prints out a string. The address of the string is initially in register `a0`, and is moved to another register we have named `s`.

Strings are represented by the address of memory containing the text. The end of the string is indicated by a null byte.

The line

```
        s         =    s0;
```

specifies that the identifier `s` really means `s0` (the name of a register).

The line

```
        mov       $a0, $s;                     //   Pointer to char in string
```

says copy the contents of register `a0` into register `s`.

The line

```
             ldbu      $a0, ($s);              //   Get character
```

says load the byte at the address indicated by `s`, into register `a0`.

The line

```
             beq       $a0, end;               //   Break if at end of string
```

says if `a0` is equal to 0, branch to the label "`end`" (in other words, set the program counter to the address corresponding to the label "`end`"). This is based on the convention of using a zero byte to indicate the end of a text string.

The line

```
                  bsr       Sys.putChar.enter; //   Print char
```

says invoke the function "`putChar`". (`bsr` stands for branch to subroutine. Subroutine is another name for function, procedure, or method.) "`Sys.putChar.enter`" is in fact a function that prints the character passed in register `a0`.

The line

```
             addq      $s,  1;                 //   Increment pointer
```

says add 1 to the register `s`. In other words, increment the pointer, to point to the next character.

The line

```
             br        while;
```

says branch back to the start of the loop, to process the next character.

Again, the names given to the labels ("`while`", "`do`", "`end`") are chosen by the programmer. I chose them because I am implementing a while loop, and these names make the structure of the program clearer.

Altogether, the program goes through a loop, printing the character pointed to, and incrementing the pointer. It terminates when it finds a null byte.

4. Instruction Syntax

What is the syntax of instructions written in assembly language? Consider a couple of examples:

```
addq      $t0, $t1, $t3;
subq      $t0, 23,  $t4;
```

We start with a symbolic opcode (operation code), representing the operation to be performed. This is used to specify the opcode field in the assembled instruction, together with the function code, for operate instructions. For example, the above instructions have symbolic opcodes "`addq`" and "`subq`". These instructions happen to translate into an actual opcode of 0x10, with function codes 0x20 and 0x29, respectively.

We follow the opcodes by a comma separated sequence of operands, then a semicolon.

In the above examples, we add the contents of registers `$t0` and `$t1` together, and put the answer in register `$t3`, and we subtract the value `23` (decimal) from the contents of register `$t0`, and put the answer in register `$t4`.

There are five kinds of operands. The opcode determines the number and kind of legal operands.

- Register.

  The operand represents a source or destination register. It is written as "`$register`", for example, `$a0`, `$v0`.

- Unsigned 8 bit constant.

  The second operand of an integer operate instruction can be of this form. The constant is written directly, without any additional annotation, for example `23` in the above `subq` instruction.

- Memory address.

  The operand represents a memory address, computed as a displacement (offset) from a base register. It is written as "`displacement($register)`", and means the `displacement` + contents of integer register `$register`. The displacement is a signed 16 bit integer. The displacement may be omitted if it is 0, allowing the notation "`($register)`". If the register is `$zero` (register 31), which always contains 0, then the operand can be written with just the displacement. For example, we can write `24($t0)`. The notation `($t0)` is an abbreviation for `0($t0)`, and the notation `1234` is an abbreviation for `1234($zero)`. Displacement operands can only be used in load and store instructions.

- Branch destination.

  The operand represents a destination address for a branch instruction. It is written directly as the destination address, but is stored as a displacement from the address just after the branch instruction. The last two bits of the displacement are not stored in the instruction, because they are always 0.

- Unsigned 26 bit constant.

  The operand of a special instruction is of this form. For example, `CALL_PAL_CALLSYS` in a `call_pal` instruction.

§4.1 Integer operate instructions

Integer operate instructions are used to perform operations on values in integer registers.

Instructions corresponding to integer operate instructions have three operands.

An integer operate instruction of the form

```
    opcode $regA, $regB, $regC;
```

means

```
    intReg[ regC ] = intReg[ regA ] operation intReg[ regB ];
```

for the specified operation. For example, "`subq $t0, $t2, $t4;`" means

```
    intReg[ t4 ] = intReg[ t0 ] - intReg[ t2 ];
```

An operate instruction of the form

```
    opcode $regA, constant, $regC;
```

means

```
    intReg[ regC ] = intReg[ regA ] operation constant;
```

for the specified operation. For example, "`subq $t0, 3, $t4;`" means

```
    intReg[ t4 ] = intReg[ t0 ] - 3;
```

We can omit the destination operand `$regC`, if it is the same as the first source. The assembler puts it in for us. for example "`addq $t1, 1;`" means

```
    intReg[ t1 ] = intReg[ t1 ] + 1;
```

Floating point operate instructions are similar, except all registers are floating point registers, and a constant is not allowed for the second operand.

Some integer operate instructions for performing arithmetic are "`addq`", "`subq`", "`mulq`", "`divq`", "`modq`", to evaluate expressions involving +, -, *, /, and %. The last two do not exist on the real machine, but do on the simulator. There is also a "`umulh`"instruction, that computes the high quadword of the product of two quadwords, to permit the full 128 bit result to be computed. It is useful for performing arithmetic on large values.

Some integer operate instructions for performing boolean computations are "`and`", "`bic`" (bit clear), "`bis`" (bit set) or "`or`", "`eqv`" (equivalent) or "`xornot`" (exclusive or not), "`ornot`", "`xor`" (exclusive or), corresponding to &, & ~, |, ^ ~, | ~, ^. For example, "`bic $1, $2, $3;`" means

```
    intReg[ 3 ] = intReg[ 1 ] & ~ intReg[ 2 ];
```

These instructions interpret the data as bit patterns of boolean flags, rather than integers.

There are also three shift instructions, "`sll`" (shift left logical), "`sra`" (shift right arithmetic), and "`srl`" (shift right logical), corresponding to <<, >> and >>>. these instructions are used to shift the bit patterns left and right. The shift logical instructions fill the vacated bits with 0, while the shift right arithmetic instruction fills the vacated bits with the sign bit. These instructions can be used to extract fields out of a bit pattern, and interpret them as either unsigned or signed numbers. They also provide a cheap way to multiply or divide by a power of 2.

Exercise DATAREP1

Suppose we have the following values in registers.

```
$t0  0x0000000000000000
$t1  0xffffffffffffff93
$t2  0x123456789abcdef0
$t3  0x8888888888888888
$t4  0x7777777777777777
$t5  0x0000000000000000
$t6  0x0000000000000000
$t7  0x0000000000000000
$t8  0x0000000000000000
```

How will the registers change after executing the instructions

```
subq $t0, 1;
addq $t1, 0x94;
sll  $t2, 7;
srl  $t3, 1,   $t5;
```

Instruction Syntax

```
sra  $t3, 1,   $t6;
srl  $t4, 1,   $t7;
sra  $t4, 1,   $t8;
```

§4.2 Load and store instructions

To operate on memory values, we must first load the source data from memory, perform the computation, then store the result back in memory.

Load and store instructions have the form

```
      opcode $regA, displacement($regB);
```

All involve computing a memory address `displacement + intReg[regB]`.

Integer load instructions load an appropriate number of bytes starting at the specified address, and store them in `intReg[regA]`. For example, `ldq` (load quadword) loads the 8 bytes corresponding to a quadword, starting at the memory address, into register `intReg[regA]`. The instruction `ldbu` (load byte unsigned) loads a single byte from the memory address, into the low byte of register `intReg[regA]`, making the high 7 bytes zero.

Integer store instructions store an appropriate number of bytes from register `intReg[regA]` to the memory starting at the specified address. For example, `stq` (store quadword) stores all 8 bytes from register `intReg[regA]` corresponding to a quadword, into memory starting at the memory address. The instruction `stb` (store byte) stores the low byte of register `intReg[regA]`, into memory at the memory address.

The default on the Alpha is to store data in memory in little endian format.

For example the ldq instruction performs the following algorithm:

```
     Quadword address = displacement + intReg[ regB ];
     Quadword data = 0;
     for ( int i = 0; i < 8; i ++ )
          data |= memory[ address + i ] << ( 8 * i );
     intReg[ regA ] = data;
```

There is also a `lda` (load address) instruction, that loads the address into the register, rather than the contents of the memory at the address.

```
     Quadword address = displacement + intReg[ regB ];
     intReg[ regA ] = address;
```

Really the load address instruction is like an `add` instruction with a constant, except that the constant is a 16 bit signed value, rather than an 8 bit unsigned value. It is often used when passing reference parameters to functions.

Floating point load and store instructions are similar to integer load and store instructions, except regA is a floating point register.

Exercise DATAREP2

Suppose we have memory

```
0x1000000 0x123456789abcdef0
0x1000008 0x0000000000000000
0x1000010 0x0000000000000000
```

How will the registers and memory change after executing the instructions

```
ldiq $t0, 0x1000000;
ldq  $t1, ($t0);
stb  $t1, 8($t0);
ldbu $t2, 2($t0);
sll  $t2, 56,  $t3;
sra  $t3, 56,  $t4;
stq  $t4, 16($t0);
```

Instruction Syntax

§4.3 Unconditional branch and jump instructions

Branch instructions are used to change the flow of control in a program.

The unconditional branch instruction has the form
```
     br   destination;
```
It is used to branch to the specified destination address (usually a label).

Essentially it corresponds to
```
     programCounter = destination;
```
The unconditional jump instruction has the form
```
     jmp  ($reg);
```
It is, used to jump to an address when the destination has to be computed at run time. It is often used to implement switch statements.

Essentially it corresponds to
```
     programCounter = intReg[ reg ];
```

§4.4 Subroutine invocation and return instructions

The `bsr` (branch to subroutine) instruction has the form
```
     bsr  destination;
```
It is used to branch to code for a function (subroutine, function, procedure and method are words that mean essentially the same thing). It remembers the address just after bsr instruction (in a register called the return address register) so that it is possible to return to this address.

Essentially it corresponds to
```
     intReg[ ra ] = programCounter;
     programCounter = destination;
```
The jsr (jump to subroutine) instruction has the form
```
     jsr  ($reg);
```
It is, used to invoke a function when the destination has to be computed at run time. It is often used to implement the invocation of instance methods in object oriented languages.

Essentially it corresponds to
```
     intReg[ ra ] = programCounter;
     programCounter = intReg[ reg ];
```
There is a matching instruction to return from a function, namely the ret instruction. It has no operands.
```
     ret;
```
It restores the program counter to its previous value.

Essentially it corresponds to
```
     programCounter = intReg[ ra ];
```
we will deal with function invocations later in more detail.

§4.5 Conditional branch instructions

Conditional branch instructions have the form
```
     opcode $regA, destination;
```
The instruction checks the value of the register, and branches to a destination only if the register satisfies some condition.
```
     if ( relation holds for intReg[ regA ] )
          programCounter = destination;
```

Instruction Syntax

Opcodes can be "`beq`" (branch if equal to 0), "`bne`" (branch if not equal to zero), "`blt`" (branch if less than 0), "`ble`" (branch if less than or equal to 0), "`bgt`" (branch if greater than 0), "`bge`" (branch if greater than or equal to 0), "`blbs`" (branch if the low bit is set), and "`blbc`" (branch if the low bit is clear). The low bit means the units bit of an integer value. A bit is said to be set if it is 1, and clear if it is 0. My personal programming style is to use `blbs` and `blbc` for testing Boolean values, rather than `bne` and `beq`.

Exercise DATAREP3

Suppose memory `Label` contains the quad value `0x123456789abcdef0`.

What are the values in the individual bytes starting at the address `Label`?

What do the instructions

```
     ldiq $t0, 8;
     ldiq $t1, 0;
     ldiq $t2, Label;
loop:
     beq  $t0, end;
     ldbu $t3, ($t2);
     addq $t2, 1;
     subq $t0, 1;
     sll  $t1, 8;
     or   $t1, $t3;
     br   loop;
end:
```

achieve, for an arbitrary value stored at address `Label`?

§4.6 Compare instructions

A class of integer operate instruction we have not mentioned is the class of compare instructions. These instructions are used to compare two arithmetic operands and create a boolean value. They correspond to the relational operators ==, <, <=. The opcodes are "`cmpeq`" (compare equal), for testing equality, "`cmplt`" (compare signed less than), "`cmple`" (compare signed less than or equal), "`cmpult`" (compare unsigned less than), "`cmpule`" (compare unsigned less than or equal). These instructions compare the values of the first and second operands, and put the boolean result in the destination register (1 for true, 0 for false). These instructions can be combined with either a "`blbs`" or "`blbc`" instruction to branch to a destination if a condition holds between two arithmetic values. Because the ordering of unsigned and signed values is different, we need different instructions for performing unsigned and signed comparisons.

§4.7 Conditional move instructions

Another class of integer operate instruction we have not mentioned is the class of conditional move instructions. They could be replaced by sequences of other instructions, but they provide a concise and efficient implementation in some special situations. They are unusual in that they may or may not modify the destination register. These instructions compare the value of `intReg[ regA ]` with 0, and either do nothing, if a relation does not hold, or copy the second operand into `intReg[ regC ]`.

An integer operate instruction of the form

```
     opcode $regA, $regB, $regC;
```

means

```
     if ( relation holds for intReg[ regA ] )
          intReg[ regC ] = intReg[ regB ];
```

An operate instruction of the form

```
     opcode $regA, constant, $regC;
```

Instruction Syntax

means

```
     if ( relation holds for intReg[ regA ] )
          intReg[ regC ] = constant;
```

Opcodes can be "`cmoveq`" (conditional move if equal to 0), "`cmovne`" (conditional move if not equal to zero), "`cmovlt`" (conditional move if less than 0), "`cmovle`" (conditional move if less than or equal to 0), "`cmovgt`" (conditional move if greater than 0), "`cmovge`" (conditional move if greater than or equal to 0), "`cmovlbs`" (conditional move if the low bit is set), and "`cmovlbc`" (conditional move if the low bit is clear).

§4.8 Special instructions

Special instructions have the form

```
     opcode constant;
```

The only special instruction we will use directly is the `call_pal` instruction, with the operand `CALL_PAL_CALLSYS`. Essentially this instruction, with this operand causes the invocation of a function in the operating system. Additional information is passed in registers `a0`, `a1`, ... `a5`, to specify the request (in `a0`) and parameters to the request. The operating system passes the result back in register `v0`.

§4.9 Pseudoinstructions

It is also possible to write some things that look like real instructions, but are not. They are what are called pseudoinstructions. The assembler translates them into different real instructions. They are recognised by the assembler to make assembly language easier to write and more readable.

The `ldiq` (load immediate quadword) pseudoinstruction has the form

```
      ldiq $regA, constant;
```

This pseudoinstruction has the effect of loading the constant into the register. In fact it is translated into a `ldq` instruction of the form

```
      ldq $regA, displacement($gp);
```

The assembler creates a table, containing all the constants, and makes the `gp` (global pointer) register point to this table. The constant can be accessed as a displacement from this register.

For small constants, there are other ways of loading the constant into a register. The `clr`, `mov` or `negq` pseudoinstructions can be used to load zero, an 8 bit positive, or 8 bit negative constant into a register. The `lda` (load address) instruction can be used to load a 16 bit signed constant into a register, by making the base register, register 31.

The `clr` (clear) pseudoinstruction has the form

```
     clr $regC;
```

and clears the specified register. It translates into

```
     bis  $zero, $zero, $regC;
```

The `mov` (move) pseudoinstruction has the form

```
     mov $regB, $regC;
```

or

```
     mov constant, $regC;
```

and moves the contents of register `regB` or an 8 bit unsigned constant into register `regC`. It translates into

```
     addq $zero, $regB, $regC;
```

or

```
     addq $zero, constant, $regC;
```

The `negq` (negate) pseudoinstruction has the form

```
     negq $regB, $regC;
```

or

```
     negq constant, $regC;
```

and negates the contents of register `regB` or an 8 bit unsigned constant and stores the result in register `regC`. It translates into

```
     subq $zero, $regB, $regC;
```

or

```
     subq $zero, constant, $regC;
```

5. Use of registers

To make life safer for all concerned, there are conventions that drivers of motor vehicles are meant to satisfy: Drive on the left hand side of the road, don't exceed the speed limit, give way to traffic crossing from the right, stop at a red light, etc. There are similar conventions for the use of registers on the Alpha. It is possible to write code that does not satisfy these conventions, but you are likely to get into trouble if you do, especially if the program is the work of more than one programmer.

Most of the conventions related to registers are to do with how they are used with function calls, and we will deal with these conventions more fully at that time. However, lets give a rough description now.

\$t0-\$t11	Temporary registers, used to hold temporary values, when evaluating expressions, etc.
\$s0-\$s5	Saved registers, used to hold the values of local variables in functions.
\$a0-\$a5	Argument registers, used to pass parameters to functions.
\$v0	Value register, used to return the result of a function.
\$ra	Return address register, used to hold the return address of a function.
\$gp	Global pointer register, used to point to the table of constants.
\$sp	Stack pointer register, used to point to the top of the stack used to allocate space for functions.
\$zero	Zero register, that always contains the value zero. Attempting to write to this register has no effect.

It is imortant to realise that on return from a function, the values of temporary registers, argument registers, and the \$v0 and \$ra registers may have been altered. Thus you cannot keep important data in these registers across function invocations. Only the saved registers, stack pointer register and global pointer register can be guaranteed to have the same value on return from a function that they had before the invocation.

6. Programs, sections and blocks

§6.1 Overall structure

An assembly language program starts with an optional entry point specification (default, start of the code section), followed by a sequence of import statements, sections and blocks.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/string.lib.s";
import "../IMPORT/number.lib.s";
import "../IMPORT/io.lib.s";

//   char buffer[ BUFFERSIZE + 1 ];
//   void main() {
//        while ( TRUE ) {
//             print( "Type some input: " );
//             if ( readline( buffer, BUFFERSIZE ) == null )
//                  break;
//             print( "The input was: " );
//             print( buffer );
//             newline();
//             }
//        print( "Bye!" );
//        exit( 0 );
//        }
block main uses proc {
     abs {
          BUFFERSIZE     =    200;
          } abs
     const {
          align;
     message1:
          asciiz    "Type some input: ";
          align;
     message2:
          asciiz    "The input was: ";
          align;
     message3:
          asciiz    "Bye!\n";
          } const
     data {
          align;
     buffer:
          byte [ BUFFERSIZE + 1 ];
          } data
     code {
     public enter:
               {
          loop:
               ldiq $a0, message1;
               bsr       IO.print.enter;
               ldiq $a0,      buffer;
               ldiq $a1, BUFFERSIZE;
               bsr       IO.readLine.enter;
               beq       $v0, end;
               ldiq $a0, message2;
               bsr       IO.print.enter;
               ldiq $a0, buffer;
               bsr       IO.print.enter;
               bsr       IO.newline.enter;
               br        loop;
          end:
               }
               {
               ldiq $a0, message3;
               bsr       IO.print.enter;
               }
               {
               clr       $a0;
               bsr       Sys.exit.enter;
               }
          } code
     } block main
```

So in the above program, the entry point is the label enter, within the block main.

An absolute section contains declarations of symbolic names for constants. Using symbolic names provides a way of making our programs easy to read. For example, we can declare symbolic names for registers.

A code section is used to specify instructions to execute.

A constant section is used to specify the data for string constants, etc.

A data section is used to specify the space for global variables. It is often used for global arrays.

A local section is used to specify the offsets for local variables for functions, fields of a record (class), etc. Basically it is used to specify the offsets of data in any kind of compound data structure.

A block is a named compound object, composed of sections, sub-blocks, etc. A block is often used to contain all the code for a function.

§6.2 Allocating space for global variables

So long as our program is small, we can use the saved registers to store the values of variables. However, registers can only be used to contain simple values, such as integers, characters, boolean values, etc. Arrays and strings are too big to be stored in a register, and have to be stored in memory. Also, it is fairly easy to run out of registers to use for simple variables, because there are only 6 saved registers. Space for string constants can be allocated in the constant section. Space for variables and arrays can be allocated in the data section. To allocate space, we need an alignment statement, a label to name the memory, then a memory allocation statement. We can initialise memory, by specifying a data type, followed by the initial value, then a ";".

```
     const {
          align;
     message1:
          asciiz    "Type some input: ";
          align;
     message2:
          asciiz    "The input was: ";
          } const
```

Data types can be keywords such as `byte`, `ubyte`, `quad`, `ascii`, `asciiz`, etc, to allocate space for a signed byte, unsigned byte, signed quadword, unterminated ASCII string, null terminated ASCII string, etc.

Apart from the data types corresponding to strings, memory allocation instructions allocate the appropriate amount of memory in the relevant section (1 byte for `byte` and `ubyte`, 2 bytes for `word` and `uword`, 4 bytes for `long` and `ulong`, 8 bytes for `quad` and `uquad`, 4 bytes for `float`, 8 bytes for `double`). The difference between the signed and unsigned variants is to do with checking the value is in range. For example `byte` requires a value that is between -0x80 and +0x7f, while `ubyte` requires a value that is between 0 and +0xff. In fact there is no checking for `quad` and `uquad`.

For `ascii` the number of bytes allocated is equal to the length of the string, and the contents is the data within the string. The `asciiz` directive is similar, except an extra zero byte is allocated and added on the end.

If we miss out the initial value, we get data that is initially zero.

```
     data {
     c:   quad;
     d:   quad;
          } data
```

We can allocate blocks of memory, by declaring an array:

```
     data {
          align;
     buffer:
          byte [ BUFFERSIZE + 1 ];
          } data
```

Uninitialised memory statements usually only occur within a data or local section.

Alignment statements can be used to round the current address up to a multiple of the size of a specified type. This is needed because data has to be aligned appropriately, for it to be accessed. Generally, it is a good idea to align data labels to quadwords, no matter what the size of the data. If labels are not at least aligned to longwords, then the memory display in the simulator will be confused.

Exercise DATAREP4

Suppose we have the following alpha assembly language

```
data {
          align;
     message:
          asciiz    "0x12\n";
     value:
          quad 0x123456789a;
     } data
```

Indicate the contents of each byte of memory in hexadecimal.

0x1000000		0x1000008	
0x1000001		0x1000009	
0x1000002		0x100000a	
0x1000003		0x100000b	
0x1000004		0x100000c	
0x1000005		0x100000d	
0x1000006		0x100000e	
0x1000007		0x100000f	

The label `message`, is at address 0x1000000

Exercise DATAREP5

Suppose we have the following alpha assembly language

```
data {
     value1:
          quad -3;        //   Note this is negative!
     value2:
          quad 1046;      //   Note this is decimal!
     } data
```

Indicate the contents of each byte of memory in **hexadecimal**.

0x1000000		0x1000008	
0x1000001		0x1000009	
0x1000002		0x100000a	
0x1000003		0x100000b	
0x1000004		0x100000c	
0x1000005		0x100000d	
0x1000006		0x100000e	
0x1000007		0x100000f	

Assume the label `value1`, is at address 0x1000000, and integers are represented in little-endian format.

Exercise TESTPROG_BIN

Show the values of memory and registers used by the following program, each time the program reaches the labels `showData1` and `showData2`.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";

block main uses proc {
     abs {
          c              =    s0;
          value          =    s1;
          textPtr        =    s2;
          } abs
     data {
          align;
     output:
          byte[ 8 ];
     endOutput:
          byte 0;
          } data
     code {
     public enter:
          ldiq     $value,       13;       //   Decimal 13.
          ldiq     $textPtr,     endOutput;
          clr      $c;
               {
          do:
          showData1:
               and      $value,       1,        $c;
               srl      $value,       1;
               addq     $c,           '0';
               subq     $textPtr,     1;
               stb      $c,           ($textPtr);
          while:
               bne      $value,       do;
          end:
               }
          showData2:
               {
          while:
               ldbu     $a0,      ($textPtr);
               beq      $a0,      end;
          do:
               bsr      Sys.putChar.enter;
               addq     $textPtr, 1;
               br       while;
          end:
               }
          clr      $a0;
          bsr      Sys.exit.enter;
          } code
     } block main
```

pc					
$c					
$value					
$textPtr					

0x1000004					
0x1000005					
0x1000006					
0x1000007					
0x1000008					

What is the output from the above program?

What does the above program achieve in general, with the number 13 replaced by an an arbitrary number?

§6.3 Creating code for simple statements and expressions

Suppose we want to increment a simple variable "a", stored in memory. We have to write something like the following:

```
ldiq $t0, a;        //   Get the address of a
ldq  $t1, ($t0);    //   Get the value of a
addq $t1, 1;        //   Increment the value
stq  $t1, ($t0);    //   Store the result back in a
```

So you can see that even the most trivial of high level statements becomes rather involved in assembly language. Of course if we used a register to store the value of the variable "a", we could have done it in one instruction.

The code to load a constant value into register tempReg is just

```
ldiq $tempReg, constantValue;
```

The code to load the address of a global variable "a" into register tempReg is just

```
ldiq $tempReg, a;
```

The code to load the value of a global variable "a" into register tempReg is just

```
ldiq $tempReg, a;
ldq  $tempReg, ($tempReg);
```

To generate code for an assignment statement "lhs = rhs;", we have to perform the following algorithm

```
Generate code to evaluate the rhs into t0;
Generate code to evaluate the address of the lhs into t1;
stq  $t0, ($t1);    //   Store the value back in the lhs address
```

The above code can be improved if we can access the lhs address as a displacement from a register.

To generate code to evaluate an expression corresponding to a binary expression "leftOpd opr rightOpd", into register tempReg, we have perform the following recursive algorithm

```
Generate code to evaluate leftOpd into tempReg;
Generate code to evaluate rightOpd into tempReg + 1;
opcode    $tempReg, $tempReg+1;    //   perform the operation
```

For example, to evaluate a * b + c * d, we would write

```
     ldiq $t0, a;
     ldq  $t0, ($t0);
     ldiq $t1, b;
     ldq  $t1, ($t1);
     mulq $t0, $t1;
     ldiq $t1, c;
     ldq  $t1, ($t1);
     ldiq $t2, d;
     ldq  $t2, ($t2);
     mulq $t1, $t2;
     addq $t0, $t1;
```

So long as we don't run out of registers, this recursive algorithm is straightforward. The algorithm for unary operators is similar.

Data is often packed together within a single quadword or longword. For example, the opcode, register numbers and displacement for a load or store instruction are packed together as fields within a longword. How can we extract the data out? We can use a left shift instruction to shift the data to the high end of a quadword (deleting the information to the left of the field), then a right shift instruction to shift the data to the low end of the quadword (deleting the information to the right of the field, and putting the data in the right place). We use a srl (shift right logical) instruction if we want to interpret the data as an unsigned number, and a sra (shift right arithmetic) instruction if we want to interpret the data as a signed number.

For example, to extract the opcode, regA, regB and displacement fields of a load/store instruction, we could write:

```
     data {
          align;
     instruction:
          long;
          align;
     opcode:
          quad;
          align;
     regA:
          quad;
          align;
     regB:
          quad;
          align;
     displacement:
          quad;
          } data
     code {
          ...
          ldiq $t0, instruction;
          ldl  $t0, ($t0);

          sll  $t0, 64-32,    $t1;
          srl  $t1, 64-32+26, $t1;
          ldiq $t2, opcode;
          stq  $t1, ($t2);

          sll  $t0, 64-26,    $t1;
          srl  $t1, 64-26+21, $t1;
          ldiq $t2, regA;
          stq  $t1, ($t2);
          sll  $t0, 64-21,    $t1;
          srl  $t1, 64-21+16, $t1;
          ldiq $t2, regB;
          stq  $t1, ($t2);

          sll  $t0, 64-16,    $t1;
          sra  $t1, 64-16+0,  $t1;
          ldiq $t2, displacement;
          stq  $t1, ($t2);
          ...
          } code
```

§6.4 Creating control structures

It is possible to build if statements and loops out of branch statements.

To create an if statement corresponding to

```
     if ( condition )
          statement1;
     else
          statement2;
```

we write

```
     {
     if:
          Generate code to evaluate the condition,
          and branch to the label "then" if the condition is true
          or "else" if the condition is false;
     then:
          Generate code for statement1;
          br   end;
     else:
          Generate code for statement2;
     end:
     }
```

The label names are arbitrary, but using the names "if", "then", "else" and "end" gives the appearance of a high level control structure.

For example, assuming all variables are stored in memory,

```
     if ( a != 0 )

          count = count + 1;

     else

          count = count - 1;
```

translates into

```
     {
     if:
          ldiq $t0, a;
          ldq  $t0, ($t0);
          beq  $t0, else;
     then:
          ldiq $t0, count;
          ldq  $t1, ($t0);
          addq $t1, 1;
          stq  $t1, ($t0);
          br   end;
     else:
          ldiq $t0, count;
          ldq  $t1, ($t0);
          subq $t1, 1;
          stq  $t1, ($t0);
     end:
     }
```

To create an if statement corresponding to

```
     if ( condition )
          statement1;
```

we write

```
     {
     if:
          Generate code to evaluate the condition,
          and branch to the label "then" if the condition is true
          or "end" if the condition is false;
     then:
          Generate code for statement1;
     end:
     }
```

To create a while statement corresponding to

```
     while ( condition )
          statement1;
```

we write

```
     {
     while:
          Generate code to evaluate the condition,
          and branch to the label "do" if the condition is true
          or "end" if the condition is false;
     do:
          Generate code for statement1;
          br   while;
     end:
     }
```

Programs, sections and blocks

For example consider

```
     result = 1;
     i = 0;
     while ( i < n ) {
          result = result * a;
          i++;
          }
```

Suppose “result”, “i”, “n” and “a” are represented by registers $result, $i, $n and $a. Then we can write

```
     mov       1,   $result;
     clr       $i;
     {
     while:
          cmplt     $i,  $n,  $t0;
          blbc      $t0, end;
     do:
          mulq      $result,  $a;
          addq      $i,  1;
          br        while;
     end:
     }
```

To create a for statement corresponding to

```
     for ( initialisation; condition; increment )
          statement1;
```

we write

```
     {
     for:
          Generate code for initialisation;
     while:
          Generate code to evaluate the condition,
          and branch to the label "do" if the condition is true
          or "end" if the condition is false;
     do:
          Generate code for statement1;
     continue:
          Generate code for the increment;
          br   while;
     end:
     }
```

For example consider what is effectively the same code as the above while loop

```
     result = 1;
     for ( i = 0; i < n; i++ )
          result = result * a;
```

Then we generate much the same code, but with a couple of additional labels, to make it look more like a for loop.

```
     mov       1,   $result;
     {
     for:
          clr       $i;
     while:
          cmplt     $i,  $n,  $t0;
          blbc      $t0, end;
     do:
          mulq      $result,  $a;
     continue:
          addq      $i,  1;
          br        while;
     end:
     }
```

Programs, sections and blocks

Break or continue statements inside the substatement should be translated into “br end;” and “br continue;” respectively.

We can also translate switch statements into assembly language. If the cases are closely packed within a limited range, we can use what is called a branch table.

```
     switch ( expr ) {
          case 0:
               stmt0;
               break;
          case 1:
               stmt1;
               break;
          case 2:
               stmt2;
               break;
          ...
          default:
               defaultStmt;
          }
```

translates into

```
     {
     switch:
          Generate code to evaluate expr into $t0;
          blt       $t0, default;
          cmple     $t0, n,   $t1;
          blbc      $t1, default;
          ldiq      $t1, branchTable;
          s8addq    $t0, $t1, $t1; //   $t1 = 8 * $t0 + $t1
          ldq       $t1, ($t1);
          jmp       ($t1);         //   Jump to the address contained in $t1
     branchTable:
          quad case0;
          quad case1;
          quad case2;
          ...
     case0:
          Generate code to evaluate stmt0;
          br   end;
     case1:
          Generate code to evaluate stmt1;
          br   end;
     case2:
          Generate code to evaluate stmt2;
          br   end;
...
     default:
          Generate code for defaultStmt;
     end:
     }
```

If the cases are sparse, we can use compare and branch instructions.

```
{
switch:
     Generate code to evaluate expr into $t0;
     cmpeq     $t0, 0,   $t1;
     blbs      $t1, case0;
     cmpeq     $t0, 1,   $t1;
     blbs      $t1, case1;
     cmpeq     $t0, 2,   $t1;
     blbs      $t1, case2;
     ...
     br        default;
case0:
     Generate code to evaluate stmt0;
     br   end;
case1:
     Generate code to evaluate stmt1;
     br   end;
case2:
     Generate code to evaluate stmt2;
     br   end;
...
default:
     Generate code for defaultStmt;
end:
}
```

What are the "{ ... }" braces for? they create a local "scope". The labels inside "{ ... }" can only be referred to inside "{ ... }", so we can use the same identifiers for labels in different control statements.

Programs, sections and blocks

7. Strings

Strings are represents as a sequence of bytes. In C, the end of a string is indicated by a zero byte. I will use the same convention.

We can create string constants by using the asciiz directive. The z in asciiz means zero byte terminated.

```
const {
     align;
message1:
     asciiz    "Type some input: ";
     align;
message2:
     asciiz    "The input was: ";
     } const
```

If we want to create new strings, we need to allocate space, using the byte directive.

```
data {
buffer:
     byte [ BUFFERSIZE + 1 ];
     } data
```

The string is limited to a maximum length of BUFFERSIZE, because the space we have allocated is BUFFERSIZE + 1 bytes (the extra byte being for the zero byte terminator). It is not easy to manage arbitrary length strings, because we then need memory management - dynamically allocating and freeing memory to hold the string.

The address of an element of a string can be accessed as the base address plus the index. To get the character at that address, we need an extra load. Note that the load instruction is ldbu, to load a byte, not a quadword.

```
ldiq $t0, buffer;
addq $i,  $t0, $t0;      //   Gives the address of the ith element.
ldbu $t0, ($t0);         //   Give the value of the ith element.
```

We can write code to read in a line of input, and store it in the buffer. Don’t worry about the code for entry to or exit from the function. just look at the code for the body of the function. If the input line is too long, the excess input is deleted. Normally, the address of the end of the text is returned. However, if end of file (represented by typing ctrl-D) is reached, null is returned instead.

```
//   char *readLine( char *s, int max ) {
//        register int i = 0;
//        register int c;
//        while ( TRUE ) {
//             c = getchar();
//             if ( c < 0 || c == '\n' )
//                  break;
//             if ( i < max )
//                  s[ i ] = c;
//             i++;
//             }
//        if ( i > max )
//             i = max;
//        s[ i ] = '\0';
//        if ( c < 0 )
//             return NULL;
//        else
//             return s + i;
//        }
//
```

```
    public block readLine uses proc {
         abs {
              s         =    s0;
              max       =    s1;
              i         =    s2;
              c         =    s3;
              } abs
         code {
         public enter:
              lda       $sp, -sav4($sp);
              stq       $ra, savRet($sp);
              stq       $s0, sav0($sp);
              stq       $s1, sav1($sp);
              stq       $s2, sav2($sp);
              stq       $s3, sav3($sp);
         body:
              mov       $a0, $s;                      //   Pointer to character
              mov       $a1, $max;                    //   Size of input buffer
              clr       $i;                           //   Count of characters read
              {
              while:
                   bsr       Sys.getChar.enter;      //   Get a char
                   mov       $v0, $c;
                   blt       $c,       end;
                   cmpeq     $c,       '\n',     $t0; //   Break if newline
                   blbs $t0, end;
              do:
                   {
                   if:
                        cmplt     $i,  $max,     $t0; //   If within buffer
                        blbc $t0, end;
                   then:
                        addq $s,       $i,  $t0;      //   Store the character
                        stb       $c,       ($t0);
                   end:
                   }
                   addq $i,  1;                       //   Increment count
                   br   while;
              end:
              }
              {
              if:
                   cmple     $i,  $max,     $t0;      //   If not within buffer
                   blbs $t0, end;
              then:
                   mov       $max,     $i;
              end:
              }
              addq $s,       $i,       $t0;
              stb       $zero,    ($t0);              //   Append null char
              {
              if:
                   bge       $c,       else;
              then:
                   clr       $v0;
                   br        end;
              else:
                   mov       $t0, $v0;
              end:
              }
```

```
          return:
               ldq       $s3, sav3($sp);
               ldq       $s2, sav2($sp);
               ldq       $s1, sav1($sp);
               ldq       $s0, sav0($sp);
               ldq       $ra, savRet($sp);
               lda       $sp, +sav4($sp);
               ret;
               } code
          } block readLine
```

The following function compares two strings. It returns a number that is < 0, == 0, > 0, if s < t, s == t, s > t, in the normal sort order.

```
     //   int compare( char *s, char *t ) {
     //        while ( *s == *t && *s != 0 ) {
     //             s++;
     //             t++;
     //             }
     //        return *s - *t;
     //        }
     public block compare uses proc {
          abs {
               s         =    a0;
               t         =    a1;
               } abs
          code {
               public enter:
               body:
                    {
                    while:
                         ldbu $t0, ($s);
                         ldbu $t1, ($t);
                         cmpeq     $t0, $t1, $t2;
                         blbc $t2, end;
                         beq       $t0, end;
                    do:
                         addq $s,       1;
                         addq $t,       1;
                    continue:
                         br        while;
                    end:
                    }
                    subq $t0, $t1, $v0;
               return:
                    ret;
               } code
          } block compare
```

The following function returns the length of a null terminated string.

```
//   int length( char *s ) {
//        int len = 0;
//        while ( *s != 0 ) {
//             len++;
//             s++;
//             }
//        return len;
//        }
public block length uses proc {
     abs {
          s         =    a0;
          len       =    s0;
          } abs
     code {
          public enter:
               lda       $sp, -sav1($sp);
               stq       $ra, savRet($sp);
               stq       $s0, sav0($sp);
          body:
               {
               for:
                    clr       $len;
               while:
                    ldbu $t0, ($s);
                    beq       $t0, end;
               do:
                    addq $len,     1;
                    addq $s,  1;
               continue:
                    br        while;
               end:
               }
               mov       $len,     $v0;
          return:
               ldq       $s0, sav0($sp);
               ldq       $ra, savRet($sp);
               lda       $sp, +sav1($sp);
               ret;
          } code
     } block length
```

The following function copies the null terminated string stored at address t to address s.

```
//   char *copy( char *s, char *t ) {
//        while ( ( *s = *t ) != 0 ) {
//             s++;
//             t++;
//             }
//        return s;
//        }
public block copy uses proc {
     abs {
          s         =    a0;
          t         =    a1;
          } abs
     code {
          public enter:
          body:
               {
               while:
                    ldbu $t0, ($t);
                    stb       $t0, ($s);
                    beq       $t0, end;
               do:
                    addq $s,       1;
                    addq $t,       1;
               continue:
                    br        while;
               end:
               }
          return:
               mov       $s,       $v0;
               ret;
          } code
     } block copy
```

Exercise UPI

Suppose we have the following Alpha assembly language program:

```
block main uses proc {
     data {
          memory1:
               quad      0x697075;
          memory2:
               quad      0;
          } data
     code {
     public enter:
          ldiq      $t0,      memory1;
          ldiq      $t1,      memory2;
          {
          while:
               ldbu      $t2, ($t0);
               beq       $t2, end;
          do:
               subq      $t2, 'a';
               addq      $t2, 'A';
               stb       $t2, ($t1);
               addq      $t0, 1;
               addq      $t1, 1;
               br        while;
          end:
          }
     showData:
          ldiq      $a0,      memory1;
          bsr       IO.print.enter;
          bsr       IO.newline.enter;
          ldiq      $a0,      memory2;
          bsr       IO.print.enter;
          bsr       IO.newline.enter;
          clr       $a0;
          bsr       Sys.exit.enter;
          } code
     } block main
```

Indicate the values in hexadecimal of the computer memory when the program reaches the label "**`showData`**". Assume "`memory1`" corresponds to address `0x1000288`, and "`memory2`" corresponds to address `0x1000290`.

0x10003f0			
0x10003f1			
0x10003f2			
0x10003f3			

Indicate the output generated by the program. Note that the bytes printed are interpreted as characters, not integers.

Exercise TESTPROG_REVERSE1

Suppose we have the following Alpha assembly language program

```
block main uses proc {
     data {        //   Address 0x1000000
          align;
     buffer:
          asciiz    "tide";
          } data
     code {
     public enter:
          ldiq $s0, buffer;
          mov  $s0, $s1;
          mov  $s0, $s2;
               {    //   loop 1
          while:
               ldbu $t0, ($s1);
               beq  $t0, end;
          do:
               addq $s1, 1;
               br   while;
          end:
               }
     showData1:
          subq $s1, 1;
               {    //   loop 2
          while:
               cmpult    $s0, $s1, $t2;
               blbc $t2, end;
          do:
               ldbu $t0, ($s0);
               ldbu $t1, ($s1);
               stb  $t0, ($s1);
               stb  $t1, ($s0);
          showData2:
          continue:
               addq $s0, 1;
               subq $s1, 1;
               br   while;
          end:
               }
     showData3:
               {    //   loop 3
          while:
               ldbu $a0, ($s2);
               beq  $a0, end;
          do:
               bsr  Sys.putChar.enter;
               addq $s2, 1;
               br   while;
          end:
               }
          mov  '\n',     $a0;
          bsr  Sys.putChar.enter;
          clr  $a0;
          bsr  Sys.exit.enter;
          } code
     } block main
```

Display the contents of registers and memory each time the program reaches, but has not executed the code at the labels **showData1**, **showData2** and **showData3**.

Indicate the value of the program counter by writing the name of the label (**showData1**, **showData2**, **showData3**) it corresponds to.

Indicate the values of registers and memory either in hexadecimal, or as an ASCII character, whichever is appropriate. The buffer starts at address `0x1000000`.

pc				
$t0				
$t1				
$t2				
$s0				
$s1				
$s2				

0x1000000				
0x1000001				
0x1000002				
0x1000003				
0x1000004				

Exercise TESTPROG_HEX

Indicate the values of registers and memory each time the program reaches the labels **showData0**, **showData1** and **showData2**.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";

block main uses proc {
     abs {
          numByte        =    2;
          numNibble      =    2 * numByte;
          i              =    s0;
          c              =    s1;
          valuePtr       =    s2;
          textPtr        =    s3;
          } abs
     data {
          align;
     text:
          byte [ numNibble ];
          align;
     value:
          word 0x7c3;
          } data
     code {
     public enter:
          ldiq     $valuePtr,    value;
          ldiq     $textPtr,     text;
     showData0:
          {
     for:
          mov       0,        $i;
     while:
          cmplt     $i,       numByte,  $t0;
          blbc      $t0,      end;
     do:
          addq      $valuePtr,    $i,  $t1;
          ldbu      $c,       ($t1);
          and       $c,       0xf, $t2;
          srl       $c,       4,        $t3;
          sll       $i,       1,        $t4;
          addq      $textPtr, $t4, $t5;
          stb       $t2,      0($t5);
          stb       $t3,      1($t5);
     showData1:
     continue:
          addq      $i,       1;
          br        while;
     end:
          }

          {
     for:
          mov       numNibble-1,        $i;
     while:
          blt       $i,       end;
     do:
          addq      $textPtr, $i,  $t6;
          ldbu      $c,       ($t6);
               {
          if:
               cmplt     $c,       10,       $t7;
               blbc      $t7,      else;
          then:
               addq      $c,       '0';
               br        end;
          else:
               addq      $c,       'a';
               subq      $c,       0xa;
          end:
               }
          mov       $c,       $a0;
          bsr       Sys.putChar.enter;
     continue:
          subq      $i,  1;
          br        while;
     end:
          }

showData2:
     bsr       Sys.exit.enter;
     } code
} block main
```

pc				
$t0				
$t1				
$t2				
$t3				
$t4				
$t5				
$i				
$c				
$valuePtr				
$textPtr				

0x1000000				
0x1000001				
0x1000002				
0x1000003				

0x1000008				
0x1000009				
0x100000a				
0x100000b				

Indicate the output generated for this specific value of 0x7c3, and the overall purpose of the program, for an arbitrary value.

Strings

Exercise TESTPROG_OCT

Indicate the values of registers and memory each time the program reaches the labels **showData0** and **showData1**.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";

block main uses proc {
     abs {
          c              =    s0;
          value          =    s1;
          textPtr        =    s2;
          } abs
     data {
     valueMem:
          quad 0x19c;
          align;
     text:
          byte [ 8 ];
          } data
     code {
     public enter:
          ldiq     $textPtr, text;
          ldiq     $t0,      valueMem;
          ldq      $value,   ($t0);
     showData0:
          stb      $zero,    ($textPtr);
               {
          do:
               and      $value,   0x7,      $t1;
               addq     $t1,      '0',      $c;
               srl      $value,   3;
               addq     $textPtr, 1;
               stb      $c,       ($textPtr);
          showData1:
          while:
               bne      $value,   do;
          end:
               }

               {
          while:
               ldbu     $a0,      ($textPtr);
               beq      $a0,      end;
          do:
               bsr      Sys.putChar.enter;
          continue:
               subq     $textPtr, 1;
               br       while;
          end:
               }

          bsr      Sys.exit.enter;
          } code
     } block main
```

pc				
$t0				
$t1				
$c				
$value				
$textPtr				

0x1000000				
0x1000001				
0x1000002				
0x1000003				

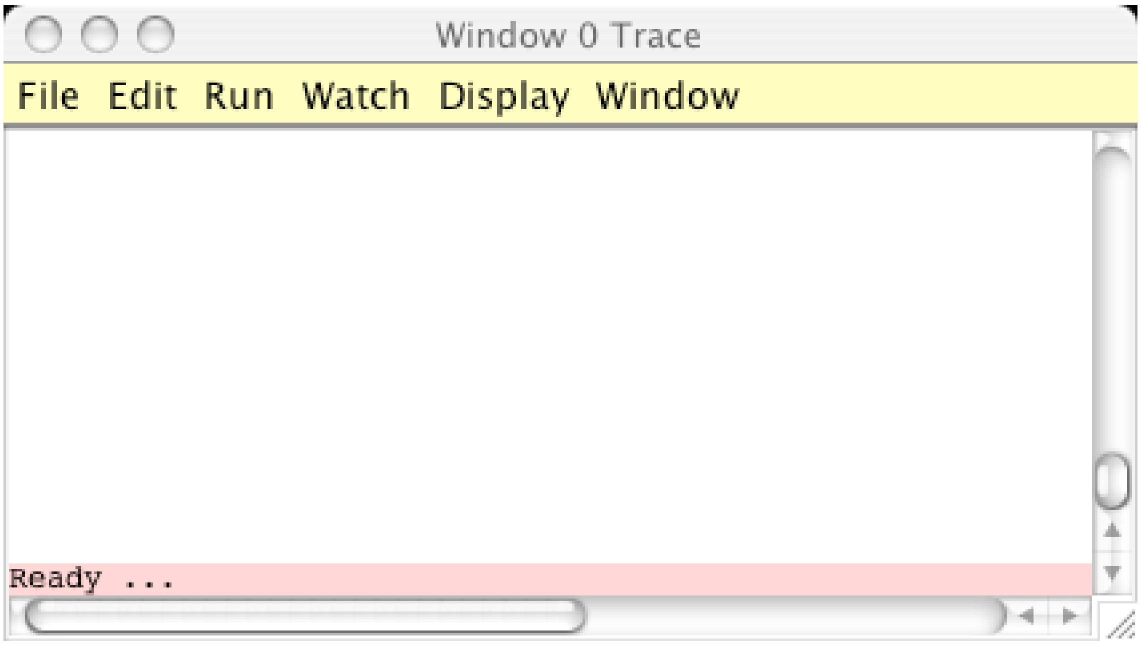
0x1000008				
0x1000009				
0x100000a				
0x100000b				

Indicate the output generated for this specific value of 0x19c, and the overall purpose of the program, for an arbitrary value.

Strings

8. Running the Alpha Simulator

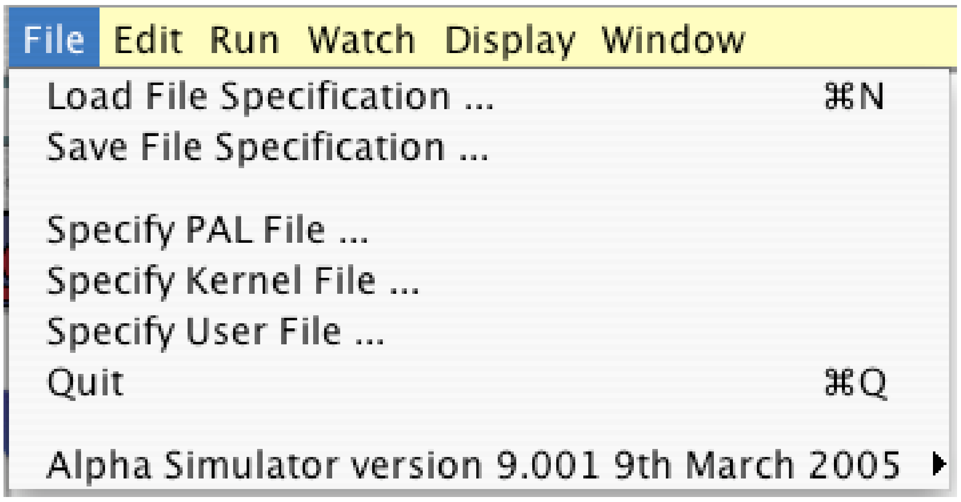
Suppose you want to run an Alpha assembly language program. First, start up the Alpha simulator application, for example by double clicking on simulator.jar. A window appears.



```
Ready ...
```

You can create additional windows, by duplicating an existing window.

§8.1 Specifying the code files to execute


File Edit Run Watch Display Window
Load File Specification ... ⌘N
Save File Specification ...
Specify PAL File ...
Specify Kernel File ...
Specify User File ...
Quit ⌘Q
Alpha Simulator version 9.001 9th March 2005

You have to specify not only your program that you want to execute, but also two other programs - the kernel code and PAL code.

The easy way of specifying the three programs is via a configuration file - basically a text file containing three lines, with the names of the three files, relative to the directory of the configuration file. The file names are in UNIX format, with path components separated by "/", and the parent directory specified by "..". It is possible to specify the three files individually, using file dialogs, but the usual way is just to specify the configuration file. The three assembly language files must have the suffixes ".pal.s", ".kernel.s" and ".user.s", for the PAL, kernel, and user files. For example, we could have a configuration file, alpha.config

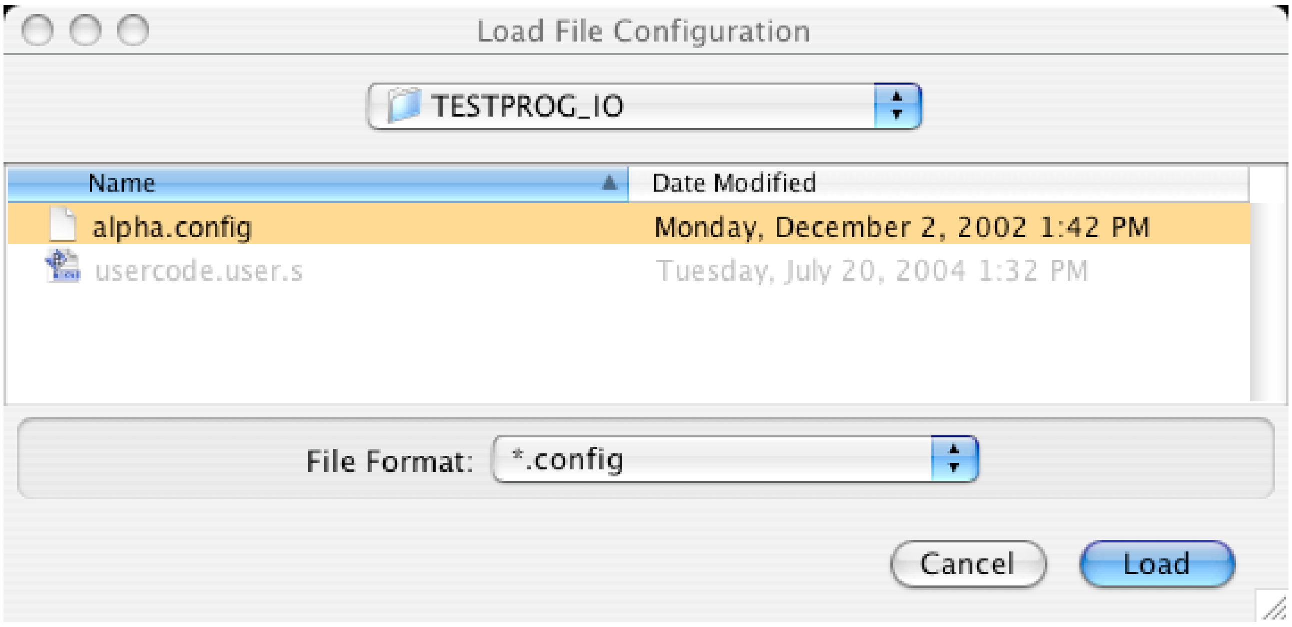
```
../SYSTEM/palcode.pal.s
../SYSTEM/kernelcode.kernel.s
usercode.user.s
```

in the same directory as the user code.

Before running a new program, you must specify the configuration file by using the

Load File Specification ... ⌘N

menu item. You can type ⌘N (Macintosh) or ctrl-N (Windows), rather than using the menu. You have to repeat this each time you want to run a different user program.



Name	Date Modified
alpha.config	Monday, December 2, 2002 1:42 PM
usercode.user.s	Tuesday, July 20, 2004 1:32 PM

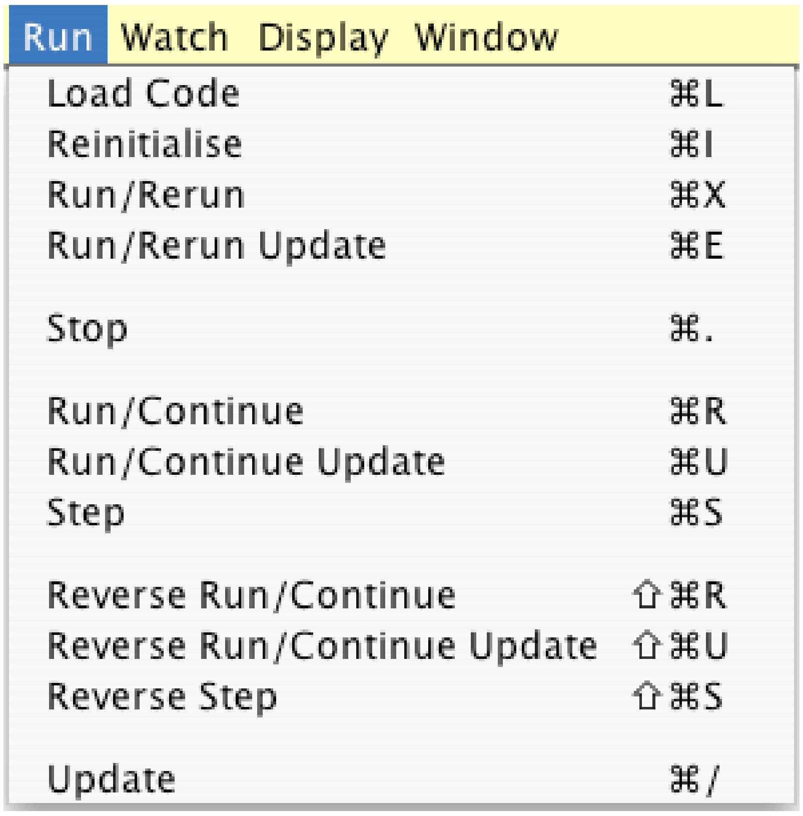
File Format: *.config

You can quit the simulator by using the

Quit ⌘Q

menu item.

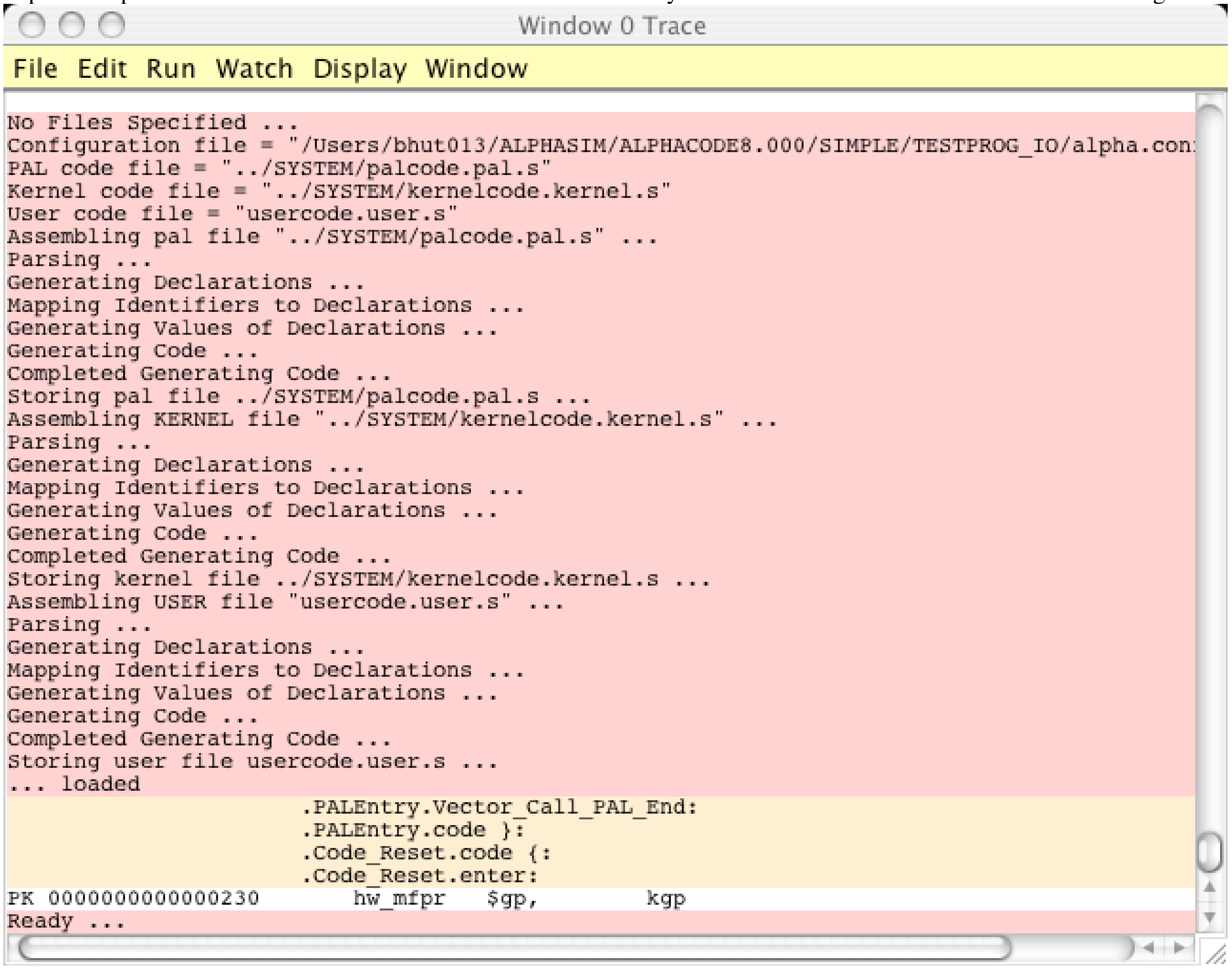
§8.2 Loading and Execution



You can assemble and load the code into the simulator's memory, by using the

Load Code ⌘L

menu item. You have to repeat this each time you modify the source code for the user program. Otherwise you will continue to run the old program.



```
No Files Specified ...
Configuration file = "/Users/bhut013/ALPHASIM/ALPHACODE8.000/SIMPLE/TESTPROG_IO/alpha.con
PAL code file = "../SYSTEM/palcode.pal.s"
Kernel code file = "../SYSTEM/kernelcode.kernel.s"
User code file = "usercode.user.s"
Assembling pal file "../SYSTEM/palcode.pal.s" ...
Parsing ...
Generating Declarations ...
Mapping Identifiers to Declarations ...
Generating Values of Declarations ...
Generating Code ...
Completed Generating Code ...
Storing pal file ../SYSTEM/palcode.pal.s ...
Assembling KERNEL file "../SYSTEM/kernelcode.kernel.s" ...
Parsing ...
Generating Declarations ...
Mapping Identifiers to Declarations ...
Generating Values of Declarations ...
Generating Code ...
Completed Generating Code ...
Storing kernel file ../SYSTEM/kernelcode.kernel.s ...
Assembling USER file "usercode.user.s" ...
Parsing ...
Generating Declarations ...
Mapping Identifiers to Declarations ...
Generating Values of Declarations ...
Generating Code ...
Completed Generating Code ...
Storing user file usercode.user.s ...
... loaded
                      .PALEntry.Vector_Call_PAL_End:
                      .PALEntry.code }:
                      .Code_Reset.code {:
                      .Code_Reset.enter:
PK 0000000000000230       hw_mfpr   $gp,        kgp
Ready ...
```

You can place watchpoints on addresses in memory and even most registers. What this means is that the simulator will stop executing if it tries to access the memory or register with a watchpoint.

You can reinitialise registers and memory by the

Reinitialise ⌘I

menu item.

You can start executing your program from the beginning, by using the

Run/Rerun ⌘X

Run/Rerun Update ⌘E

menu items. In fact the PAL initialisation code executes first, then kernel initialisation code, and finally your user code. The update option updates the trace window as it executes, but executes more slowly.

You can use Run/Rerun directly, if you have never used Load File Specification or Load Code. You will be prompted for a configuration file, and the three files will be assembled and loaded.

Your program will stop executing if it attempts to access data with an associated watchpoint, or it tries to read input and no input is available, or it reaches completion (by invoking the exit system call), or something goes wrong (an exception occurs).

You can also stop execution by using the

Stop ⌘.

menu item.

If your program stops you can resume execution from the point at which it stopped by using the

Run/Continue ⌘R

Run/Continue Update ⌘U

menu items.

It is also possible to single step through your program by using the

Step ⌘S

menu item. Windows are updated after each instruction.

The

Reverse Run/Continue ⇧⌘R

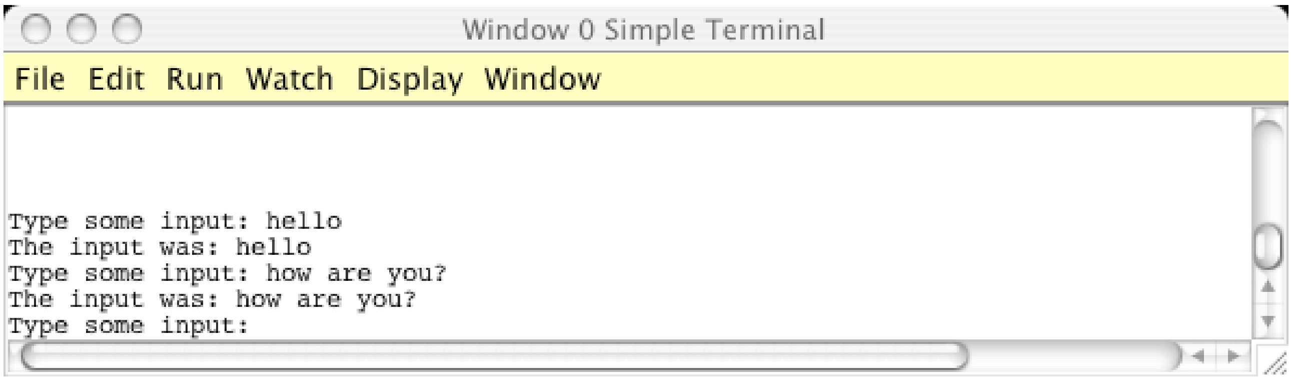
Reverse Run/Continue Update ⇧⌘U

Reverse Step ⇧⌘S

menu items can be used to run the simulator in reverse. You can only run in reverse for a few thousand instructions. Because input/output takes thousands of instructions to execute, this is not as useful as you might hope.

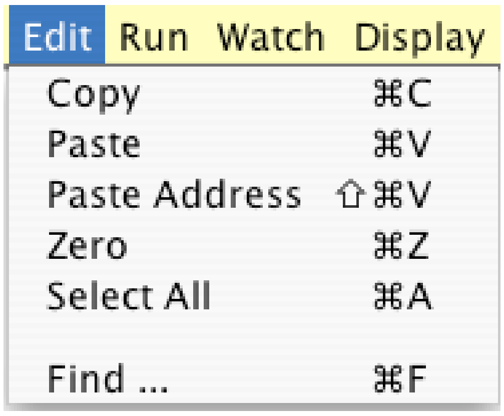
§8.3 Reading from the Simple Terminal

If you type input into the simple terminal window, it can be edited, by backspacing and retyping. Characters cannot be read until you type return.

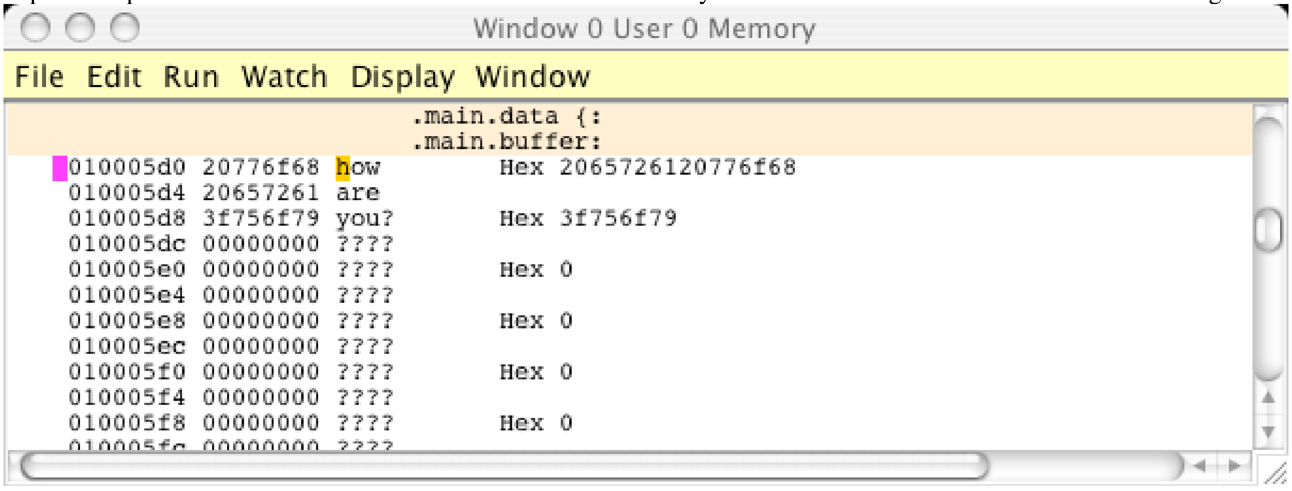


```
Type some input: hello
The input was: hello
Type some input: how are you?
The input was: how are you?
Type some input:
```

§8.4 Editing, Copying and Pasting



Text in windows can be selected by clicking, and shift clicking, or clicking and dragging. In register and memory windows, the text can be edited, by typing hexadecimal characters in the hex display, or textual characters in the text display. The cursor moves as you type.



```
                          .main.data {:
                          .main.buffer:
  010005d0 20776f68 how        Hex 2065726120776f68
  010005d4 20657261 are
  010005d8 3f756f79 you?       Hex 3f756f79
  010005dc 00000000 ????
  010005e0 00000000 ????       Hex 0
  010005e4 00000000 ????
  010005e8 00000000 ????       Hex 0
  010005ec 00000000 ????
  010005f0 00000000 ????       Hex 0
  010005f4 00000000 ????
  010005f8 00000000 ????       Hex 0
```

Whole lines of register and memory windows can be copied and pasted. Within the simulator, this copies the numerical value, not the text. If the copy size does not equal the paste size, the data is truncated, or zero extended at the high memory end.

It is possible to paste the address of the memory copied, rather than the contents.

Lines of text can be copied from the simulator, and pasted in text documents.

It is also possible to use

Save Selection ...

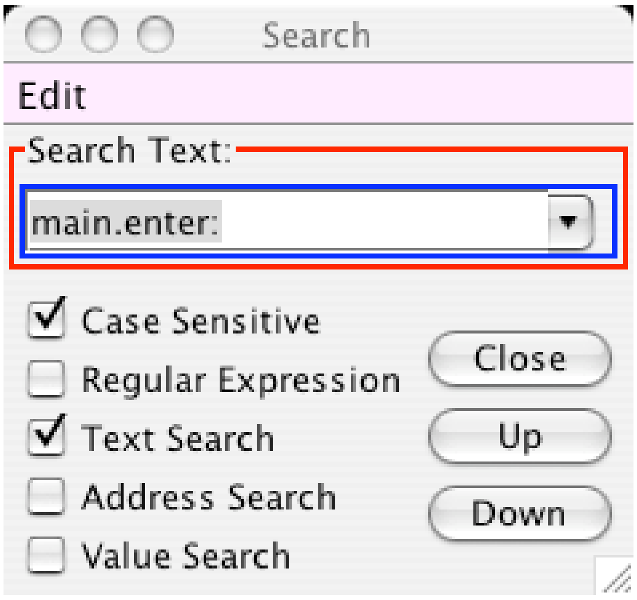
menu item in the Window menu to save a portion of a window in a text file.

§8.5 Searching

You can search for text in a window by using the

Find ... ⌘F

menu item.



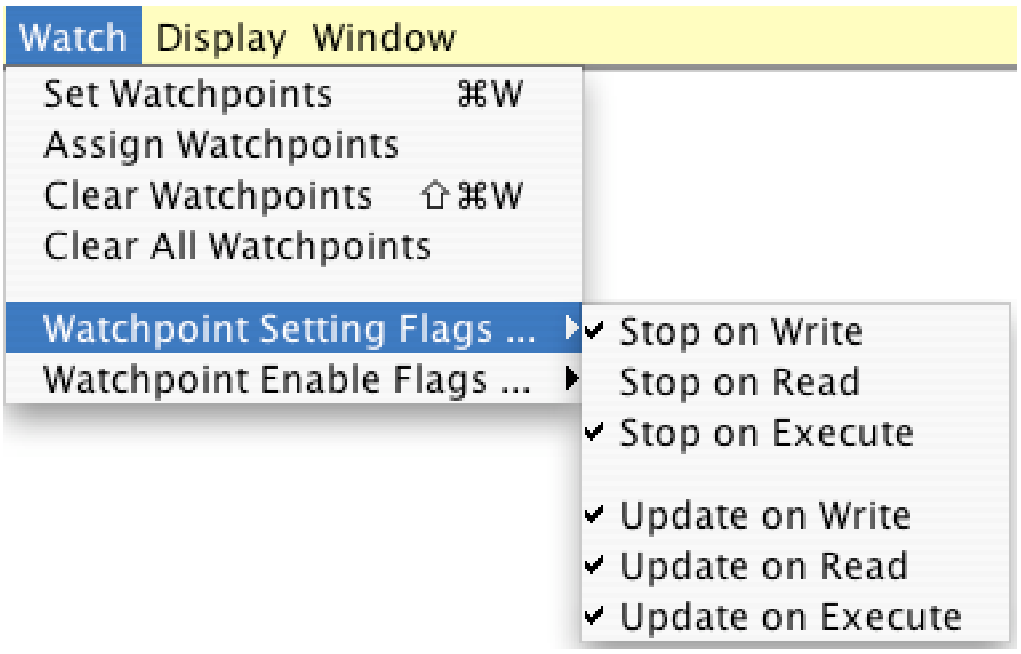
[✓ Case Sensitive]
Regular Expression
[✓ Text Search]
Address Search
Value Search

When pasting into the text field, you can paste either text, the value, or the address.

The match is usually case sensitive, but can be made insensitive. Regular expressions are permitted. You can search for either text, an address or a value.

You can select a previous search, using the menu displayed by clicking to the right of the text field.

§8.6 Setting Watchpoints



You can select a range of memory or registers, and set and clear watchpoints, using the

Set Watchpoints ⌘W
Clear Watchpoints ⇧⌘W

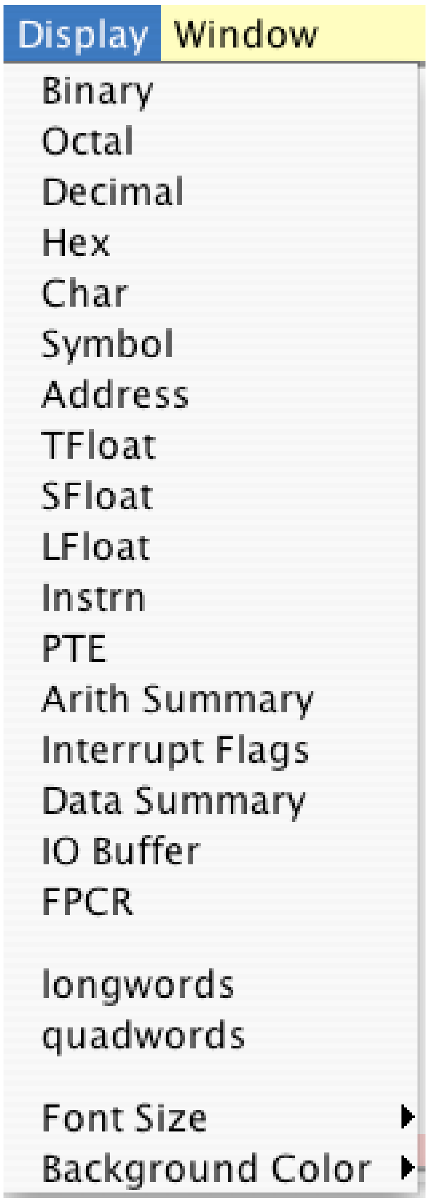
menu items.

You can also specify the watch flags that will be set, when you set a watchpoint, by specifying the

Watchpoint Setting Flags ...

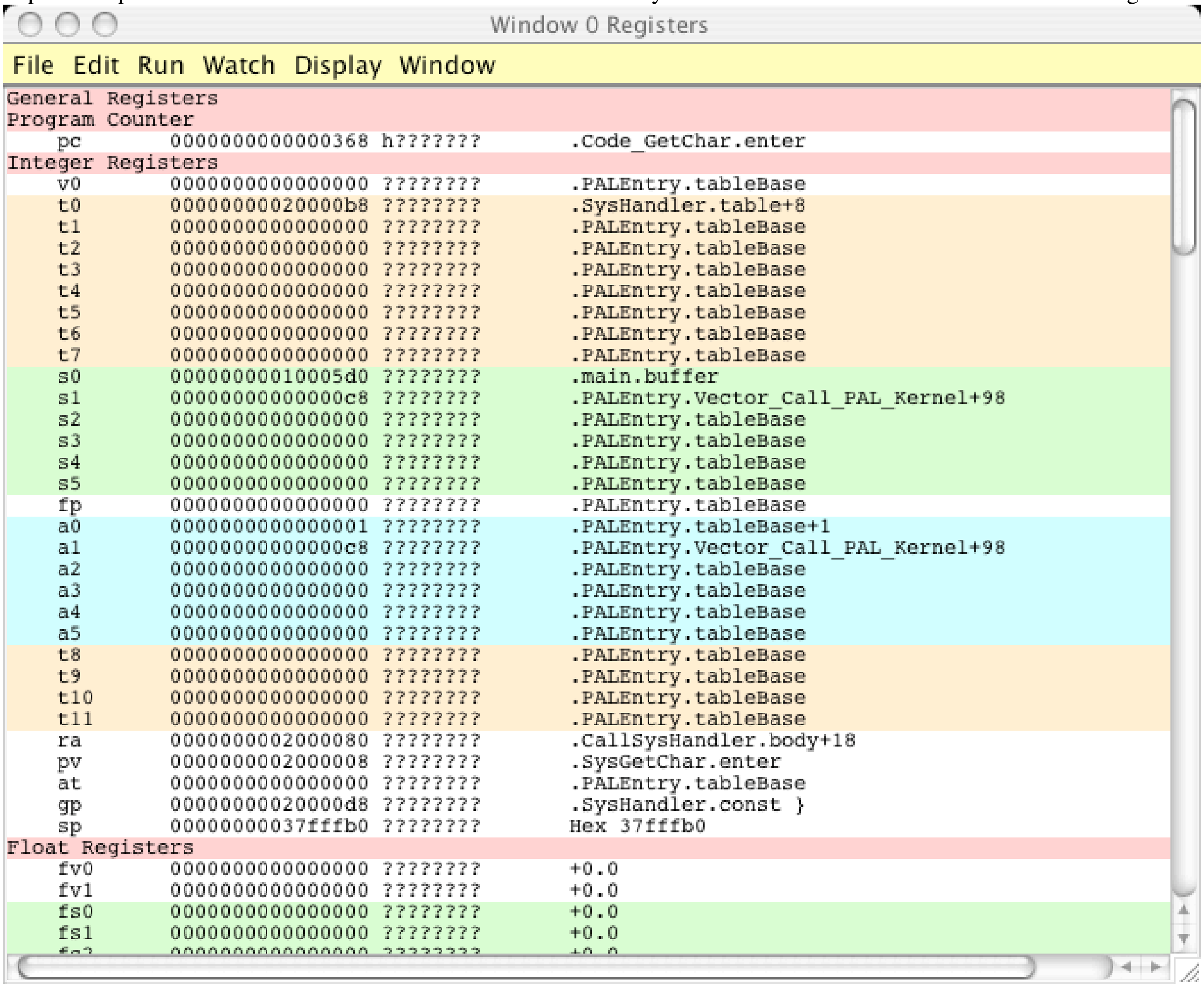
The default is to stop on write or execute, but not on read.

§8.7 Formatting



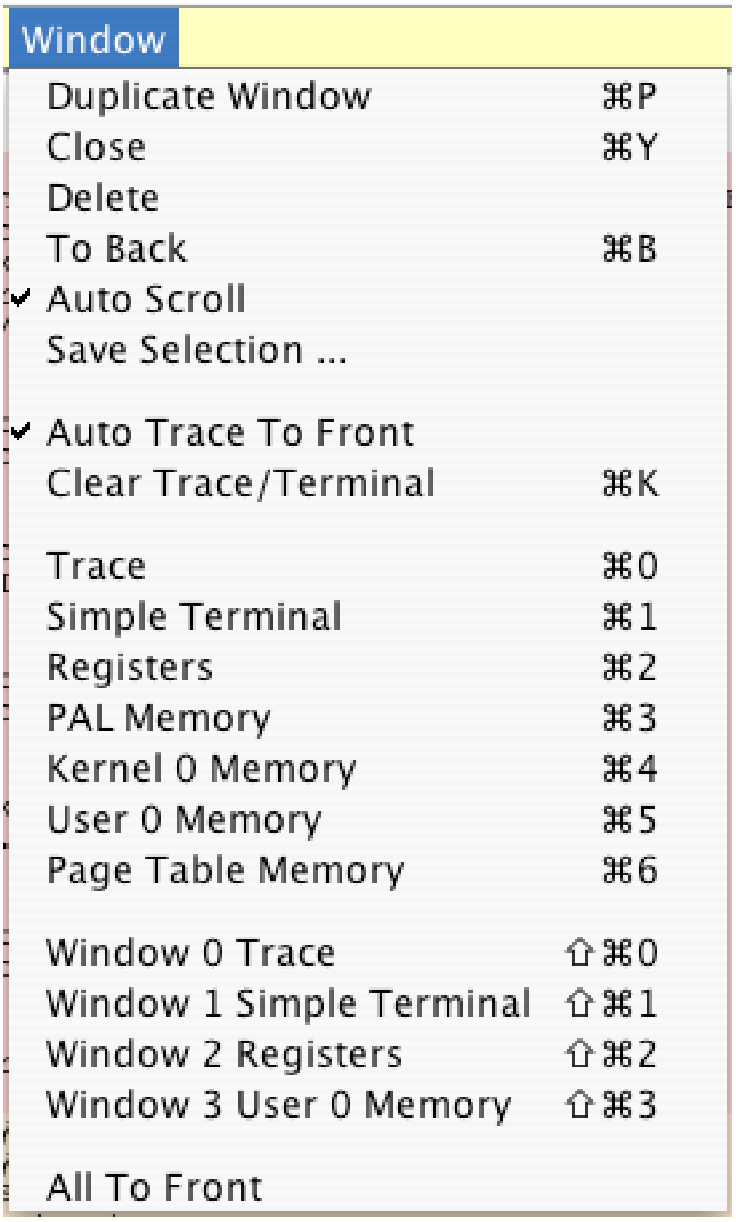
You can select a range of memory or registers and specify how the data is disassembled. For example, you can disassemble data as numbers in binary, octal, decimal or hexadecimal, as instructions, as characters, as symbolic addresses, etc. You can also specify whether memory is divided into longwords or quadwords, when disassembled.

You can also specify the font size of all text (for people with poor eyes), and the background color of lines (to highlight the lines).



```
General Registers
Program Counter
    pc        0000000000000368 h???????       .Code_GetChar.enter
Integer Registers
    v0        0000000000000000 ????????       .PALEntry.tableBase
    t0        00000000020000b8 ????????       .SysHandler.table+8
    t1        0000000000000000 ????????       .PALEntry.tableBase
    t2        0000000000000000 ????????       .PALEntry.tableBase
    t3        0000000000000000 ????????       .PALEntry.tableBase
    t4        0000000000000000 ????????       .PALEntry.tableBase
    t5        0000000000000000 ????????       .PALEntry.tableBase
    t6        0000000000000000 ????????       .PALEntry.tableBase
    t7        0000000000000000 ????????       .PALEntry.tableBase
    s0        00000000010005d0 ????????       .main.buffer
    s1        00000000000000c8 ????????       .PALEntry.Vector_Call_PAL_Kernel+98
    s2        0000000000000000 ????????       .PALEntry.tableBase
    s3        0000000000000000 ????????       .PALEntry.tableBase
    s4        0000000000000000 ????????       .PALEntry.tableBase
    s5        0000000000000000 ????????       .PALEntry.tableBase
    fp        0000000000000000 ????????       .PALEntry.tableBase
    a0        0000000000000001 ????????       .PALEntry.tableBase+1
    a1        00000000000000c8 ????????       .PALEntry.Vector_Call_PAL_Kernel+98
    a2        0000000000000000 ????????       .PALEntry.tableBase
    a3        0000000000000000 ????????       .PALEntry.tableBase
    a4        0000000000000000 ????????       .PALEntry.tableBase
    a5        0000000000000000 ????????       .PALEntry.tableBase
    t8        0000000000000000 ????????       .PALEntry.tableBase
    t9        0000000000000000 ????????       .PALEntry.tableBase
    t10       0000000000000000 ????????       .PALEntry.tableBase
    t11       0000000000000000 ????????       .PALEntry.tableBase
    ra        0000000002000080 ????????       .CallSysHandler.body+18
    pv        0000000002000008 ????????       .SysGetChar.enter
    at        0000000000000000 ????????       .PALEntry.tableBase
    gp        00000000020000d8 ????????       .SysHandler.const }
    sp        00000000037fffb0 ????????       Hex 37fffb0
Float Registers
    fv0       0000000000000000 ????????       +0.0
    fv1       0000000000000000 ????????       +0.0
    fs0       0000000000000000 ????????       +0.0
    fs1       0000000000000000 ????????       +0.0
```

§8.8 Managing Windows



The Window menu can be used to open, close, delete, and duplicate windows. You can bring windows to the front, or move them to the back. To bring a window to the front, you can use ⇧⌘0, ⇧⌘1, ⇧⌘2, etc.

To specify which panel to display a panel in the current window, you can use ⌘0, ⌘1, ⌘2, etc.

The most important panels are:

⌘0 The trace panel

A panel that displays a trace of the recently executed instructions.

⌘1 The simple terminal panel

A panel to for performing input/output.

⌘2 The register panel

A panel to display the contents of the registers.

⌘5 The user memory panel

A panel to display the contents of user memory - the user program, global data, and function stack.

There are also panels to display PAL memory, kernel memory, and the page tables.

These panels display the information needed to debug your program. Look at their contents, add watchpoints, and single step through critical code, or it will take 20 times as long to debug your program.

§8.9 What the Kernel and PAL code do

The kernel code represents a very simple operating system. The only services this simple operating system provides are reading and writing characters, and terminating the user program.

PAL code can be thought of as implementing instructions that are too complex to be implemented in hardware. We execute a call_pal instruction to execute a PAL code function. For example "call_pal CALL_PAL_CALLSYS" implements the "callsys" instruction to switch from executing user code to execute kernel (operating system) code. "call_pal CALL_PAL_RETSYS" implements the "retsys" instruction to switch from executing kernel code to execute user code.

For example, to read a single character, we might write two instructions in our user program

```
     ldiq      $a0, CALLSYS_GETCHAR;
     call_pal  CALL_PAL_CALLSYS;
```

These might assemble into the following two instructions

```
     ldq       $a0,    +0000($gp)
     call_pal 0000083
```

(The constant CALLSYS_GETCHAR is stored in the global table, and the ldiq pseudoinstruction is replaced by a ldq instruction. The constant CALL_PAL_CALLSYS is replaced by its value.

```
 U 0000000000800000   ldq    $a0,    +0000($gp)
 U 0000000000800004   call_pal 0000083
```

The first column indicates the modes the processor is executing in - PAL (P) or non-PAL ( ), user (U) or kernel (K). the second column is the value of the program counter. On the right is a disassembly of the instruction executed.

The PAL code switches over to the kernel.

```
CALL_PAL_USER 0x3 Exception
           .Code_Callsys.code {:
           .Code_Callsys.enter:
PU 0000000000000318    hw_mtpr  $t0,     temp06
PU 000000000000031c    addq   $zero,   01,     $t0
PU 0000000000000320    hw_mtpr  $zero,   currMode
PK 0000000000000324    hw_mtpr  $sp,     usp
PK 0000000000000328    hw_mfpr  $sp,     ksp
PK 000000000000032c    lda    $sp,    -0038($sp)
PK 0000000000000330    stq    $t0,    +0000($sp)
PK 0000000000000334    hw_mfpr  $t0,     intEnb
PK 0000000000000338    stq    $t1,    +0008($sp)
PK 000000000000033c    hw_mfpr  $t0,     prevPC
PK 0000000000000340    stq    $t0,    +0010($sp)
PK 0000000000000344    stq    $gp,    +0018($sp)
PK 0000000000000348    hw_mfpr  $gp,     kgp
PK 000000000000034c    addq   $zero,   05,     $t0
PK 0000000000000350    hw_mfpr  $t0,     kentry[entInt]($t0)
PK 0000000000000354    hw_mtpr  $t0,     prevPC
PK 0000000000000358    hw_mfpr  $t0,     temp06
PK 000000000000035c    hw_rei
```

The kernel code invokes a function to read a character.

```
          .CallSysHandler.code {:
```

```
          .CallSysHandler.enter:
 K 0000000002000020   lda    $sp,    -0018($sp)
 K 0000000002000024   stq    $ra,    +0000($sp)
          .CallSysHandler.body:
 K 0000000002000028   cmpult  $a0,    03,     $t0
 K 000000000200002c   blbc   $t0,    .CallSysHandler.error
 K 0000000002000030   ldq    $t0,    +0000($gp)
 K 0000000002000034   s8addq  $a0,    $t0,    $t0
 K 0000000002000038   ldq    $pv,    +0000($t0)
 K 000000000200003c   jsr    $ra,    ($pv),   0040
```

The function to read a character itself invokes PAL code to actually get the character.

```
           .SysGetChar.code {:
           .SysGetChar.enter:
 K 0000000002000008    call_pal 0000001
CALL_PAL_KERNEL 0x1 Exception
            .Code_GetChar.code {:
            .Code_GetChar.enter:
PK 0000000000000368     call_xfc XFC_GETCHAR
PK 000000000000036c     hw_rei
```

It then returns to the kernel.

```
 K 000000000200000c    ret    $zero,   ($ra),   0000
 K 0000000002000040   br    $zero,   .CallSysHandler.return
          .CallSysHandler.return:
 K 0000000002000048   ldq    $ra,    +0000($sp)
 K 000000000200004c   lda    $sp,    +0018($sp)
```

Finally, it returns to the user program via PAL code.

```
 K 0000000002000050   call_pal 000003d
CALL_PAL_KERNEL 0x3d Exception
           .Code_Retsys.code {:
           .Code_Retsys.enter:
PK 00000000000002a0    ldq    $gp,    +0018($sp)
PK 00000000000002a4    ldq    $t0,    +0010($sp)
PK 00000000000002a8    hw_mtpr  $t0,    prevPC
PK 00000000000002ac    hw_mtpr  $zero,   intEnb
PK 00000000000002b0    addq   $zero,   01,     $t0
PK 00000000000002b4    hw_mtpr  $t0,    currMode
PU 00000000000002b8    lda    $sp,    +0038($sp)
PU 00000000000002bc    hw_mtpr  $sp,    ksp
PU 00000000000002c0    hw_mfpr  $sp,    usp
PU 00000000000002c4    hw_rei
```

And after this point it will execute the user code immediately after instruction

```
     call_pal  CALL_PAL_CALLSYS;
```

So when we run the simulator, we find that as soon as we attempt to perform input or output, we end up executing large amounts of PAL code and kernel code. With a real operating system, the kernel code is far more complex. All this code appears in the trace window. We can more or less just ignore it.

9. Integer arrays

Integer arrays can be created by declaring an array of quadwords.

```
//   int array[ DATASIZE ];
     array:
          quad[ DATASIZE ];
```

The above allocates space for DATASIZE quadwords, namely 8 * DATASIZE bytes.

The address of an element of an integer array can be accessed as the base address plus 8 times the index. To get the integer at that address, we need an extra load. Note that the load instruction is ldq, to load a quadword.

```
     ldiq $t0, array;
     mulq $i,  8,   $t1;
     addq $t1, $t0, $t0;      //   Gives the address of the ith element.
     ldq  $t0, ($t0);         //   Give the value of the ith element.
```

In fact, there is special support for array indexing. The s8addq instruction is an instruction especially designed for indexing arrays of quadwords. It multiplies the first operand (the array index) by 8 (the size of a quadword), adds it to the second operand (the address of the array) and stores the result (the address of the appropriate element) in the third operand. Thus it can be used to compute the address of an array element, given the index and base address. To get the value, we then need a load instruction.

```
     ldiq      $t0, array;
     s8addq    $i,  $t0, $t0;      //   Gives the address of the ith element.
     ldq       $t0, ($t0);         //   Give the value of the ith element.
```

There is a similar instruction, s4addq, used to index arrays of longwords. Of course, the addq instruction can be used to index simple arrays of bytes. For arrays with elements of size other than 1, 4, or 8, we need an explicit multiplication of the index by the size of the elements. If the size of the elements is a power of 2, the multiplication can be done by a shift.

The following function prints out the elements of an array of quadwords.

```
//   void printArray( int[] array, int max ) {
//        int i;
//        for ( i = 0; i < max; i++ )
//             printf( "%8d", array[ i ] );
//        newline();
//        }

block printArray uses proc {
     abs {
          array     =    s0;
          max       =    s1;
          i         =    s2;
          } abs
     const {
               align;
          format:
               asciiz "%8d";
          } const
     code {
     public enter:
          lda       $sp, -sav3($sp);
          stq       $ra, savRet($sp);
          stq       $s0, sav0($sp);
          stq       $s1, sav1($sp);
          stq       $s2, sav2($sp);
     body:
          mov       $a0, $array;
          mov       $a1, $max;
          {
          for:                                //   for ( i = 0; i < max; i++ )
               clr       $i;
          while:
               cmplt     $i,       $max,     $t0;
               blbc $t0, end;
          do:
               ldiq $a0, format;
               s8addq    $i,       $array,   $t0;//   printf( "%8d", array[ i ] );
               ldq       $a1, ($t0);
               bsr       IO.printf.enter;
          continue:
               addq $i,  1;
               br        while;
          end:
          }
          bsr       IO.newline.enter;         //   newline();
     return:
          ldq       $s2, sav2($sp);
          ldq       $s1, sav1($sp);
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav3($sp);
          ret;
          } code
     } block printArray
```

The following main program reads in a sequence of decimal integers, converts them into internal form, and puts them in the array data. It then sorts them into order, using a bubble sort, and prints out the array each time the bubble sort performs a swap.

```
//   int BUFFERSIZE = 20;
//   int DATASIZE = 10;
//   char buffer[ BUFFERSIZE + 1 ];
//   int array[ DATASIZE ];
//   void main() {
//        int maxArray;
//        for ( maxArray = 0; maxArray < DATASIZE; maxArray++ ) {
//             print( "Enter a number (or return to finish): " );
//             readLine( buffer, BUFFERSIZE );
//             if ( buffer[ 0 ] == 0 )
//                  break;
//             array[ maxArray ] = Number.fromString( buffer, 10 );
//             }
//        print( "Sorting by bubble sort:\n" );
//        printArray( array, maxArray );
//        for ( int i = maxArray - 1; i > 0; --i ) {
//             for ( int j = 0; j < i; j++ ) {
//                  int temp1 = array[ j ];
//                  int temp2 = array[ j + 1 ];
//                  if ( temp1 > temp2 ) {
//                       array[ j ] = temp2;
//                       array[ j + 1 ] = temp1;
//                       printArray( array, maxArray );
//                       }
//                  }
//             newline();
//             }
//        exit( 0 );
//        }
```

```
block main uses proc {
     abs {
          BUFFERSIZE     =    20;
          DATASIZE  =    10;
          maxArray  =    s0;
          i              =    s1;
          j              =    s2;
          } abs
     const {
     message1:
          asciiz    "Enter a number (or return to finish): ";
     message2:
          asciiz    "Sorting by bubble sort:\n";
          } const
     data {
          align;
     buffer:
          byte [ BUFFERSIZE + 1 ];
     array:
          quad [ DATASIZE ];
          } data
     code {
     public enter:
          //   for ( maxArray = 0; maxArray < DATASIZE; maxArray++ ) {
          {
          for:
               clr       $maxArray;
          while:
               cmplt     $maxArray,     DATASIZE, $t0;
               blbc $t0, end;
          do:
               //   print( "Enter a number (or return to finish): " );
               ldiq $a0, message1;
               bsr       IO.print.enter;
               //   readLine( buffer, BUFFERSIZE );
               ldiq $a0, buffer;
               ldiq $a1, BUFFERSIZE;
               bsr       IO.readLine.enter;
               ldiq $t0, buffer;        //   if ( buffer[ 0 ] == 0 )
               ldbu $t0, ($t0);
               beq       $t0, end;      //        break;
               ldiq $a0, buffer;        //   array[ maxArray ] =
                                        //        Number.fromString( buffer, 10 );
               ldiq $a1, 10;
               bsr       Number.fromString.enter;
               ldiq $t0, array;
               s8addq    $maxArray,     $t0, $t0;
               stq       $v0, ($t0);
          continue:
               addq $maxArray,     1;
               br        while;         //   }
          end:
          }
          ldiq $a0, message2;           //   print( "Sorting by bubble sort:\n" );
          bsr       IO.print.enter;     //   printArray( array, maxArray );
          ldiq $a0, array;
          mov       $maxArray,     $a1;
          bsr       printArray.enter;
          //   for ( int i = maxArray - 1; i > 0; --i ) {
          {
          for:
               subq $maxArray,     1,   $i;
          while:
                beq       $i,       end;
            do:
                //   for ( int j = 0; j < i; j++ ) {
                {
                for:
                    clr       $j;
                while:
                    cmplt     $j,       $i,       $t0;
                    blbc $t0, end;
                do:
                    ldiq $t0, array;
                    s8addq    $j,       $t0, $t1;
                    addq $j,       1,        $t2;
                    s8addq    $t2, $t0, $t2;
                    ldq       $t3, ($t1);    //   int temp1 = array[ j ];
                    ldq       $t4, ($t2);    //   int temp2 = array[ j + 1 ];
                    {
                    if:                      //   if ( temp1 > temp2 ) {
                        cmple     $t3, $t4, $t5;
                        blbs $t5, end;
                    then:
                        stq       $t3, ($t2);//       array[ j ] = temp2;
                        stq       $t4, ($t1);//       array[ j + 1 ] = temp1;
                        ldiq $a0, array;     //   printArray( array, maxArray );
                        mov       $maxArray,     $a1;
                        bsr       printArray.enter;
                    end:                     //        }
                    }
                continue:
                    addq $j,       1;
                    br        while;    //        }
                end:
                }
                bsr       IO.newline.enter;  //   newline();
            continue:
                subq $i,       1;
                br        while;         //   }
            end:
            }
            clr       $a0;               //   exit( 0 );
            bsr       Sys.exit.enter;
            } code
        } block main
```

Exercise TESTPROG_MINMAX

Indicate the values of registers and memory each time the program reaches the label **`showData`**. The label `array` is at address `0x1000000`.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";

block main uses proc {
    abs {
        ARRAYSIZE      =     5;
        i              =    s0;
        arrayPtr       =    s1;
        a              =    s2;
        b              =    s3;
        x              =    s4;
        } abs
```

Integer arrays

```
data {
array:
     quad 0x3;
     quad 0x7;
     quad 0x6;
     quad 0x1;
     quad 0x9;
     } data
code {
public enter:
          {
     for:
          mov       1,   $i;
          ldiq      $arrayPtr,    array;
          ldq       $a,       ($arrayPtr);
          ldq       $b,       ($arrayPtr);
     while:
          cmplt     $i,  ARRAYSIZE,    $t0;
          blbc      $t0, end;
     do:
          s8addq    $i,  $arrayPtr,    $t1;
          ldq       $x,  ($t1);
               {
          if:
               cmple     $a,       $x,       $t2;
               blbs      $t2,      end;
          then:
               mov       $x,       $a;
          end:
               }
               {
          if:
               cmplt     $b,       $x,       $t3;
               blbc      $t3,      end;
          then:
               mov       $x,       $b;
          end:
               }
     showData:
     continue:
          addq      $i,       1;
          br        while;
     end:
          }
     clr       $a0;
     bsr       Sys.exit.enter;
     } code
} block main
```

$t0				
$t1				
$t2				
$t3				
$i				
$arrayPtr				
$a				
$b				
$x				

0x1000000				
0x1000008				
0x1000010				
0x1000018				
0x1000020				

Indicate the overall purpose of the program, for arbitrary data in the array.

Exercise TESTPROG_COMPACT

Suppose we have the following Alpha assembly language program

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";

data {
     arrayStart:   //   Address 0x1000000
          quad 0x0;
          quad 0x2;
          quad 0x7;
          quad 0x0;
          quad 0x6;
     arrayFinish:
     } data

block doIt uses proc {
     abs {
          start     =    a0;
          finish    =    a1;
          cond =    t0;
          value     =    t1;
          p    =    t2;
          q    =    t3;
          } abs
     code {
          public enter:
               {
               for:
                    mov  $start,   $p;
                    mov  $start,   $q;
               while:
                    cmplt     $p,  $finish,  $cond;
               showData:
                    blbc $cond,    end;
               do:
                    {
                    if:
                         ldq  $value,   ($p);
```

```
                        beq  $value,   end;
                   then:
                        stq  $zero,    ($p);
                        stq  $value,   ($q);
                        addq $q,  8;
                   end:
                   }
              continue:
                   addq $p,  8;
                   br   while;
              end:
              }
              ret;
         } code
    } block doIt

//  int main() {
//       doIt( arrayStart, arrayFinish );
//       exit( 0 );
//       }
block main uses proc {
    code {
    public enter:
         ldiq $a0, arrayStart;
         ldiq $a1, arrayFinish;
         bsr  doIt.enter;
         clr  $a0;
         bsr  Sys.exit.enter;
         } code
    } block main
```

Display the contents of registers and memory each time the program reaches, but has not executed the code at the label **showData**.

Indicate the values of registers and memory in hexadecimal.

The label **arrayStart** corresponds to address `0x1000000`.

$start						
$finish						
$cond						
$value						
$p						
$q						

0x1000000						
0x1000008						
0x1000010						
0x1000018						
0x1000020						

Indicate what the whole program achieves, in general, for arbitrary data in the array.

10. Writing and Debugging Assembly language Programs

How can we go about writing and debugging assembly language programs? Well, unless you are writing things that can only be written in assembly language (because even high level languages such as C don't have the expressive power to represent your algorithm), then the most efficient way of developing your program is to first write it in a high level language, debug the algorithm, and then translate your high level program into assembly language. Most students don't believe me, but this really does save enormous amounts of time. It is very difficult debugging assembly language, because it is so unstructured, and there are no validity checks. Get the algorithm correct first, and the translation into assembly language is easy.

What language should you use to prototype your assembly language program? The best choice is definitely C, because it is very close to assembly language in its expressive power. You can do almost anything in C, while most high level languages have checks that stop you treating addresses as integers, storing arbitrary data at an address, etc. C is a wonderful language for doing very low level things. For example, the only languages I know that are suitable for writing a memory manager, with garbage collection, are C and assembly language, and it is 100 times easier to write it in C than assembly language.

However, if you don't know C, then use another language, such as Java, but try and use only the very low level features of the language. For example, store your strings as arrays of bytes, do your own conversion between strings and numbers, etc.

Document your assembly language for a function with the code written in a high level language.

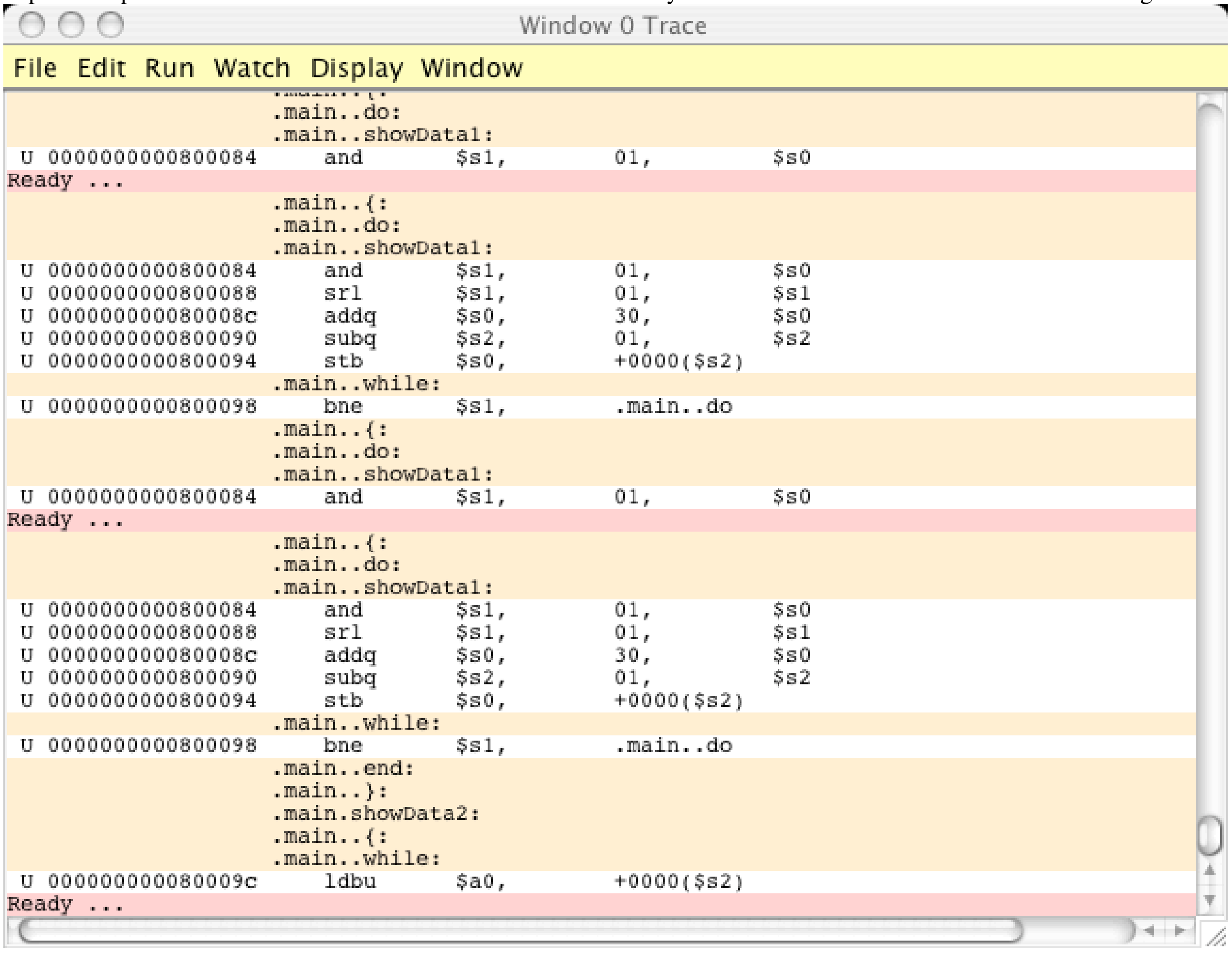
Split your assembly language program up into functions, and invoke and declare the functions using the standard function invocation conventions. Save and restore the s0, s1, s2, ... registers properly. Do not store values in temporary registers, invoke a function, then expect the values to still be there on return from the function. Always pass the parameters in a0, a1, a2, ... and return the result in v0. Make very sure that the amount of space allocated on entry to a function is identical to the amount of space deallocated on exit, and that there is sufficient space allocated. If you don't get this right, you will get some very obscure bugs, that will be very difficult to find.

Indent your assembly language, and use appropriate labels to make the control structures absolutely clear.

When you attempt to run your program, make sure that you save the assembly language text file before loading the file into the simulator. Make sure that if you change the assembly language, you reload it into the simulator. Make sure that there are no errors when you reload the program. If there are syntax errors in your program, the parser usually gives a line number. If it manages to parse everything, and then fails at the end, this tends to be a result of having unmatched braces {...}.

A general rule for debugging any program is to put lots of print statements in, so that you know what is happening. It is not so easy in assembly language, because even printing a string constant is nontrivial, printing numbers is difficult, and printing data structures is a major undertaking. However, I do supply functions to print strings, convert numbers to strings, and even implement a simple printf. If at all possible, use these. However, if you are writing assembly language that is doing tricky things, such as manipulating the stack to return from multiple levels of function invocation, you might find it difficult to use print statements.

To make life easy, the simulator provides you with a trace of the instructions executed.

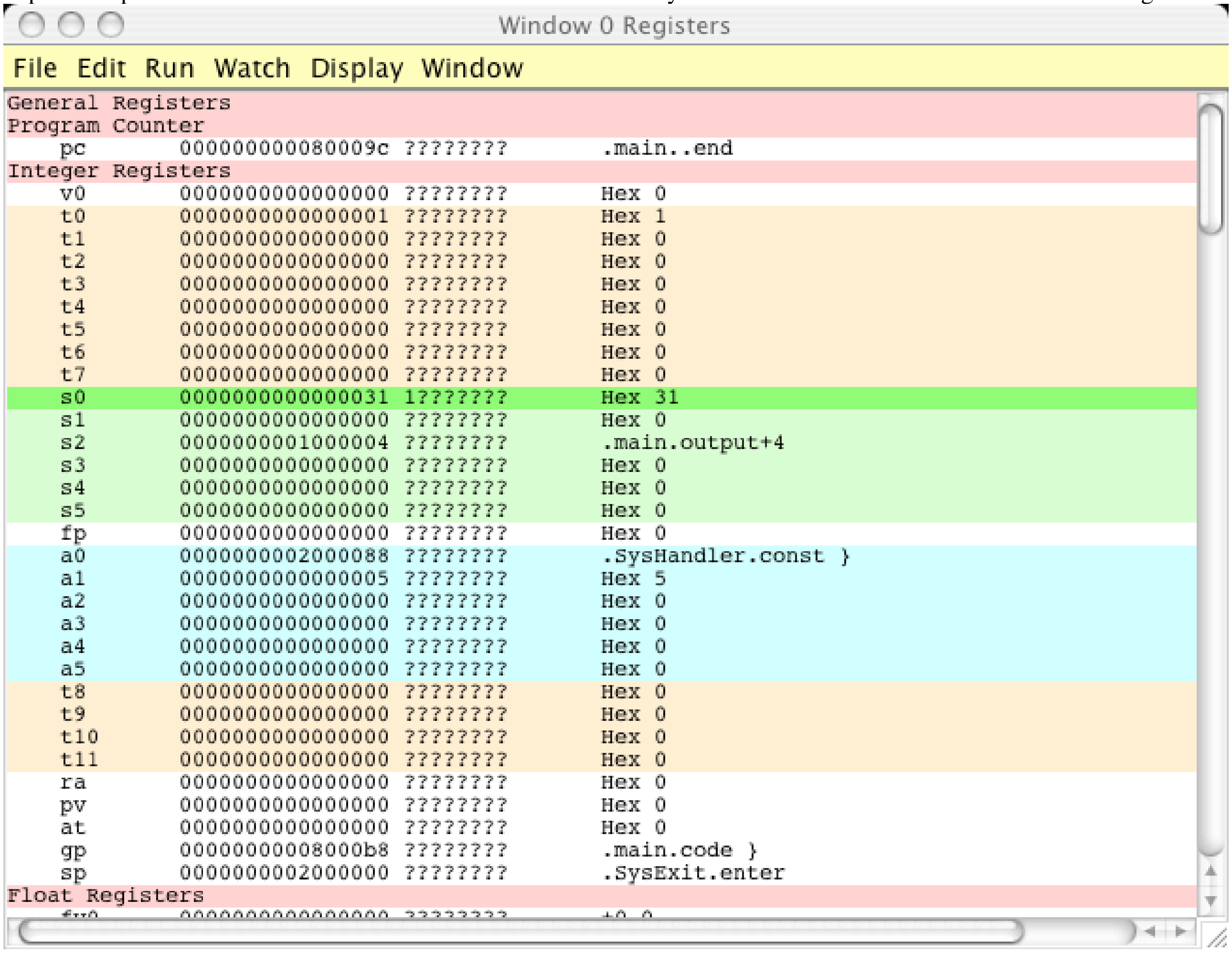


```
                          .main..do:
                          .main..showData1:
 U 0000000000800084           and         $s1,          01,           $s0
Ready ...
                          .main..{:
                          .main..do:
                          .main..showData1:
 U 0000000000800084           and         $s1,          01,           $s0
 U 0000000000800088           srl         $s1,          01,           $s1
 U 000000000080008c           addq        $s0,          30,           $s0
 U 0000000000800090           subq        $s2,          01,           $s2
 U 0000000000800094           stb         $s0,          +0000($s2)
                          .main..while:
 U 0000000000800098           bne         $s1,          .main..do
                          .main..{:
                          .main..do:
                          .main..showData1:
 U 0000000000800084           and         $s1,          01,           $s0
Ready ...
                          .main..{:
                          .main..do:
                          .main..showData1:
 U 0000000000800084           and         $s1,          01,           $s0
 U 0000000000800088           srl         $s1,          01,           $s1
 U 000000000080008c           addq        $s0,          30,           $s0
 U 0000000000800090           subq        $s2,          01,           $s2
 U 0000000000800094           stb         $s0,          +0000($s2)
                          .main..while:
 U 0000000000800098           bne         $s1,          .main..do
                          .main..end:
                          .main..}:
                          .main.showData2:
                          .main..{:
                          .main..while:
 U 000000000080009c           ldbu        $a0,          +0000($s2)
Ready ...
```

If your program stops at some point, due to an exception occurring, look at the last instruction it tried to execute. If it was a load or store instruction, and it generated a DTB_MISS_NATIVE or D_FAULT exception, then maybe the value of the base register used to compute the memory address is wrong. See what value it has. See what code modified its value, and whether it makes sense.

When your program stops at some point, you have to map the last instruction in the trace window back to a line in your assembly language. Sometimes it is obvious, because the labels are also shown in the trace window. Sometimes it is not so clear. The trace window shows the value of the program counter. Look for this address in the user memory window, and you will see the disassembly of your program, including the relevant labels. It is much easier to see how this relates to your original assembly language. However, you will still find that pseudoinstructions have been replaced by different real instructions. For example, “ldiq $reg, XXX” will appear as “ldq $reg, YYY($gp);”.

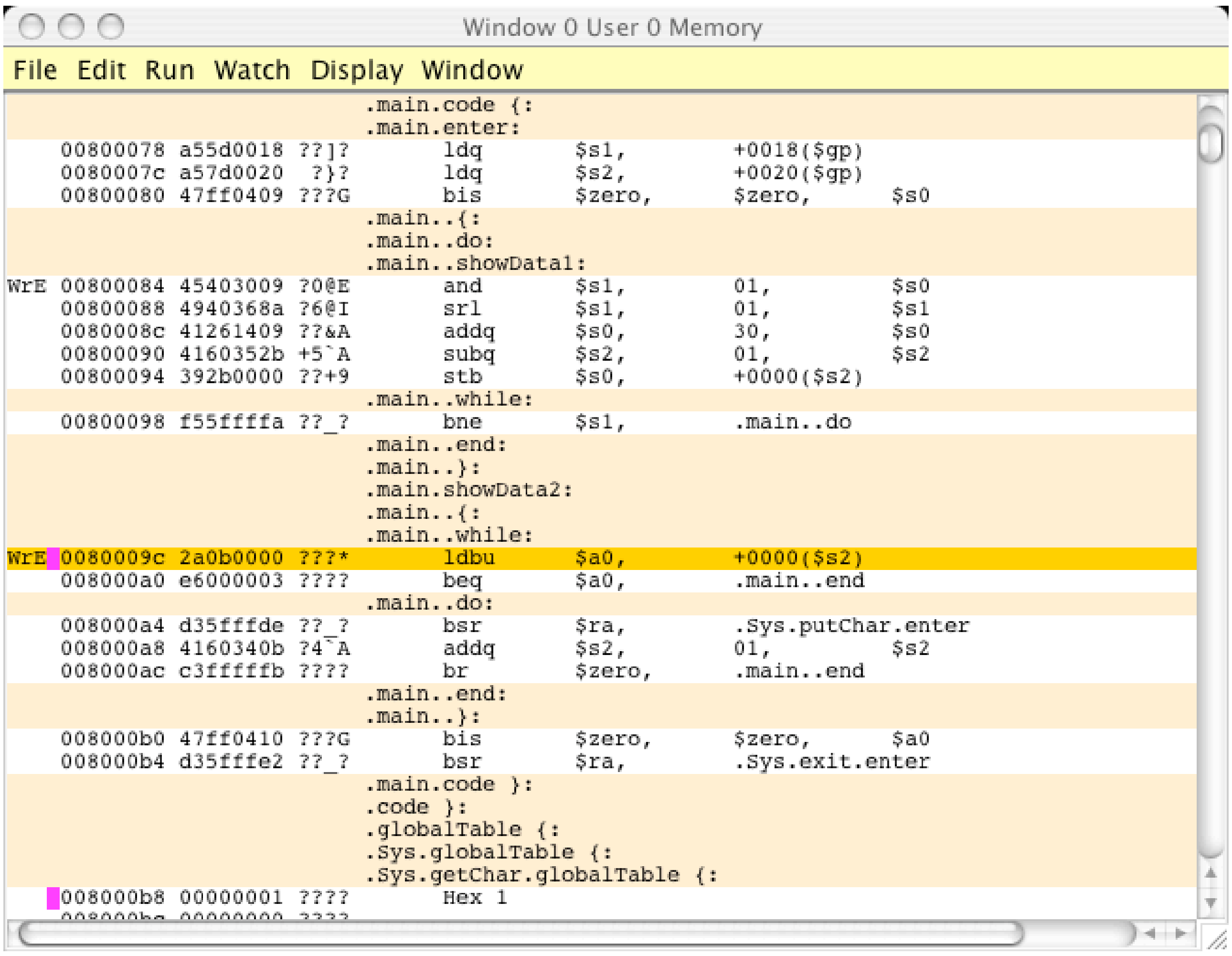
The simulator permits you to view and change the contents of registers and memory. For each register, the register window contains the name of the register, its value in hexadecimal and as characters, and a disassembly in whatever format you choose.



```
General Registers
Program Counter
    pc        000000000080009c ????????      .main..end
Integer Registers
    v0        0000000000000000 ????????      Hex 0
    t0        0000000000000001 ????????      Hex 1
    t1        0000000000000000 ????????      Hex 0
    t2        0000000000000000 ????????      Hex 0
    t3        0000000000000000 ????????      Hex 0
    t4        0000000000000000 ????????      Hex 0
    t5        0000000000000000 ????????      Hex 0
    t6        0000000000000000 ????????      Hex 0
    t7        0000000000000000 ????????      Hex 0
    s0        0000000000000031 1???????      Hex 31
    s1        0000000000000000 ????????      Hex 0
    s2        0000000001000004 ????????      .main.output+4
    s3        0000000000000000 ????????      Hex 0
    s4        0000000000000000 ????????      Hex 0
    s5        0000000000000000 ????????      Hex 0
    fp        0000000000000000 ????????      Hex 0
    a0        0000000002000088 ????????      .SysHandler.const }
    a1        0000000000000005 ????????      Hex 5
    a2        0000000000000000 ????????      Hex 0
    a3        0000000000000000 ????????      Hex 0
    a4        0000000000000000 ????????      Hex 0
    a5        0000000000000000 ????????      Hex 0
    t8        0000000000000000 ????????      Hex 0
    t9        0000000000000000 ????????      Hex 0
    t10       0000000000000000 ????????      Hex 0
    t11       0000000000000000 ????????      Hex 0
    ra        0000000000000000 ????????      Hex 0
    pv        0000000000000000 ????????      Hex 0
    at        0000000000000000 ????????      Hex 0
    gp        00000000008000b8 ????????      .main.code }
    sp        0000000002000000 ????????      .SysExit.enter
Float Registers
```

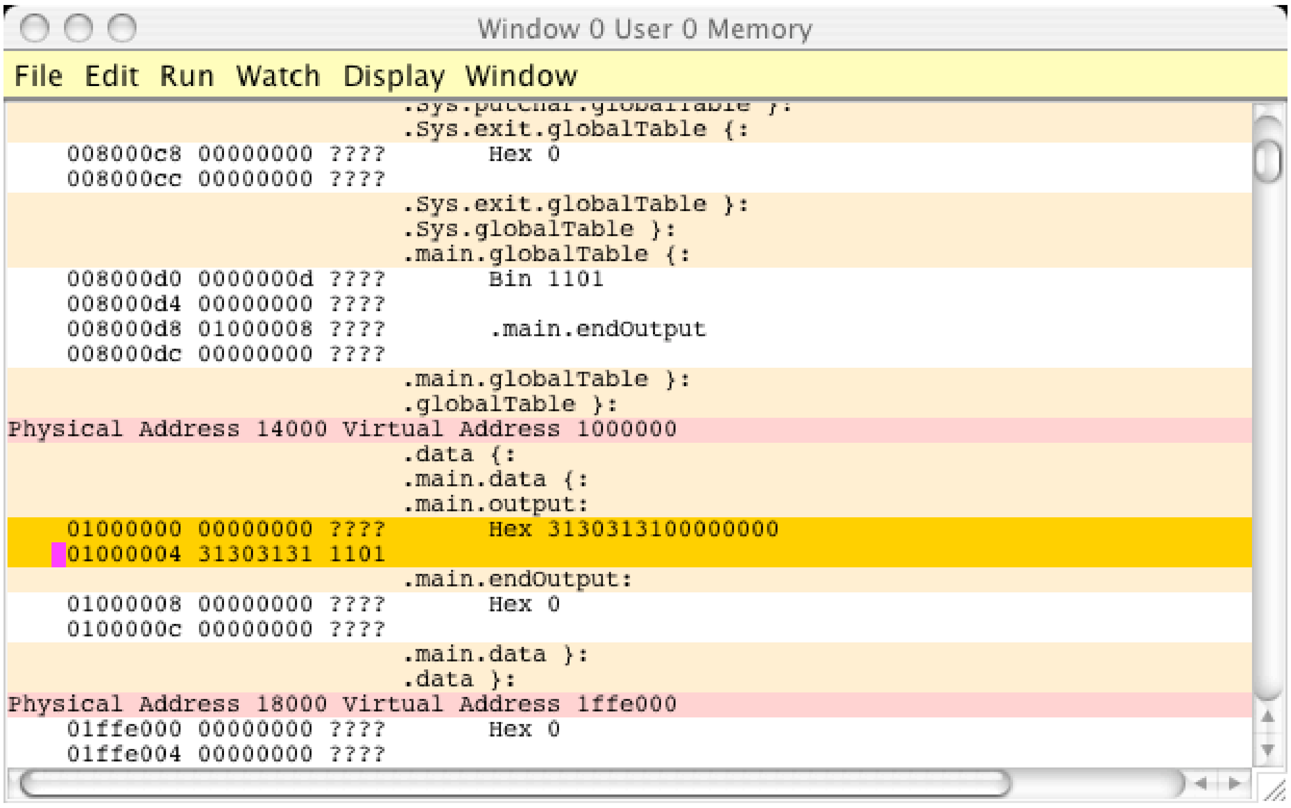
For memory, it contains the address, contents of memory in hexadecimal and as characters, and a disassembly in whatever format you choose (the default is as instructions for code, and hex for data).

The user memory window contains the user program code.



```
                            .main.code {:
                            .main.enter:
    00800078 a55d0018 ??]?         ldq       $s1,         +0018($gp)
    0080007c a57d0020  ?}?         ldq       $s2,         +0020($gp)
    00800080 47ff0409 ???G         bis       $zero,       $zero,       $s0
                            .main..{:
                            .main..do:
                            .main..showData1:
WrE 00800084 45403009 ?0@E         and       $s1,         01,          $s0
    00800088 4940368a ?6@I         srl       $s1,         01,          $s1
    0080008c 41261409 ??&A         addq      $s0,         30,          $s0
    00800090 4160352b +5`A         subq      $s2,         01,          $s2
    00800094 392b0000 ??+9         stb       $s0,         +0000($s2)
                            .main..while:
    00800098 f55ffffa ??_?         bne       $s1,         .main..do
                            .main..end:
                            .main..}:
                            .main.showData2:
                            .main..{:
                            .main..while:
WrE 0080009c 2a0b0000 ???*         ldbu      $a0,         +0000($s2)
    008000a0 e6000003 ????         beq       $a0,         .main..end
                            .main..do:
    008000a4 d35fffde ??_?         bsr       $ra,         .Sys.putChar.enter
    008000a8 4160340b ?4`A         addq      $s2,         01,          $s2
    008000ac c3fffffb ????         br        $zero,       .main..end
                            .main..end:
                            .main..}:
    008000b0 47ff0410 ???G         bis       $zero,       $zero,       $a0
    008000b4 d35fffe2 ??_?         bsr       $ra,         .Sys.exit.enter
                            .main.code }:
                            .code }:
                            .globalTable {:
                            .Sys.globalTable {:
                            .Sys.getChar.globalTable {:
    008000b8 00000001 ????         Hex 1
```

Further down are the constants, the global table, and the global variables.



```
                            .Sys.exit.globalTable {:
    008000c8 00000000 ????         Hex 0
    008000cc 00000000 ????
                            .Sys.exit.globalTable }:
                            .Sys.globalTable }:
                            .main.globalTable {:
    008000d0 0000000d ????         Bin 1101
    008000d4 00000000 ????
    008000d8 01000008 ????         .main.endOutput
    008000dc 00000000 ????
                            .main.globalTable }:
                            .globalTable }:
Physical Address 14000 Virtual Address 1000000
                            .data {:
                            .main.data {:
                            .main.output:
    01000000 00000000 ????         Hex 3130313100000000
    01000004 31303131 1101
                            .main.endOutput:
    01000008 00000000 ????         Hex 0
    0100000c 00000000 ????
                            .main.data }:
                            .data }:
Physical Address 18000 Virtual Address 1ffe000
    01ffe000 00000000 ????         Hex 0
    01ffe004 00000000 ????
```

At the bottom of the user memory window is the stack. If you are invoking functions, especially recursive ones, you might want to look at the stack.

It is possible to associate "watchpoints" with memory addresses and most registers, by selecting the data and using the Watchpoints menu. Every time your program attempts to access the data (read, write or execute, as specified) at the address, it stops, so that you can view the data, and even alter it. To alter the data, click on either the hexadecimal or character disassembly, and type appropriate characters.

If you want to check that your program is computing a value correctly, place a write watchpoint on that data, and run your program. Check the value each time it stops, then continue its execution, by selecting Run/Continue. Similarly, you can place watchpoints on instructions, so that an attempt to execute the instruction causes the program to stop.

Once a program has stopped, you can continue execution by selecting Run/Continue, or you can single step through your program by selecting Step. For small programs or portions of programs that do not attempt input/output, this works very well. However, you do not want to single step through code performing input/output and making system calls, because it is too time consuming.

Unfortunately, if you modify your program, and reload the code, all watchpoints disappear. Setting them up again can be a little time consuming. To get around this, it is possible to put watchpoints into your program in assembly language. The call_pal instruction

```
          call_pal      CALL_PAL_BPT;
```

causes your program to stop execution. A higher level way of doing this is to invoke the Sys.breakpoint function.

```
          bsr Sys.breakpoint.enter;
```

You can then restart it by selecting Run/Continue or Step.

The latest version of the simulator permits you to run the simulator backwards, so that you can see the values of registers and memory a bit earlier than the point at which you rpogram stopped. this is useful so long as the recent code does not perform input/output. The trouble is that input/output executes large numbers of instructions, and the simulator only permits you to reverse a few thousand instructions.

Another feature that can be useful is the ability to search for text in the simulator windows. For example, you can search using regular expressions in the latest version of the simulator.

11. Function invocations and declarations

§11.1 Overview

How do we generate assembly language for function invocations and declarations? While it is not the full story, it roughly amounts to the following.

For the invocation of a function:

- Generate code to evaluate the parameters and store them in a standard place. On the Alpha, the first six parameters are stored in the argument registers a0, a1, ... a5.
- Generate a function invocation instruction that saves the program counter in a standard place and sets the program counter to the address of the start of the function. On the Alpha, the bsr (branch to subroutine) instruction is used to invoke a function. The bsr instruction usually saves the old program counter in the ra (return address) register.
- Generate code to use the result of the function, assuming the result of the function is stored in a standard place. On the Alpha, functions return their result in the v0 register.

For example, to execute "x = f(5, 8, 2);", we might write:

```
     mov  5,   $a0;
     mov  8,   $a1;
     mov  2,   $a2;
     bsr  f.enter;
     ldiq $t0, x;
     stq  $v0, ($t0);
```

Control is passed to the function by the execution of the bsr instruction. On completion of the execution of the code for the function, control will be passed back to just after the bsr instruction.

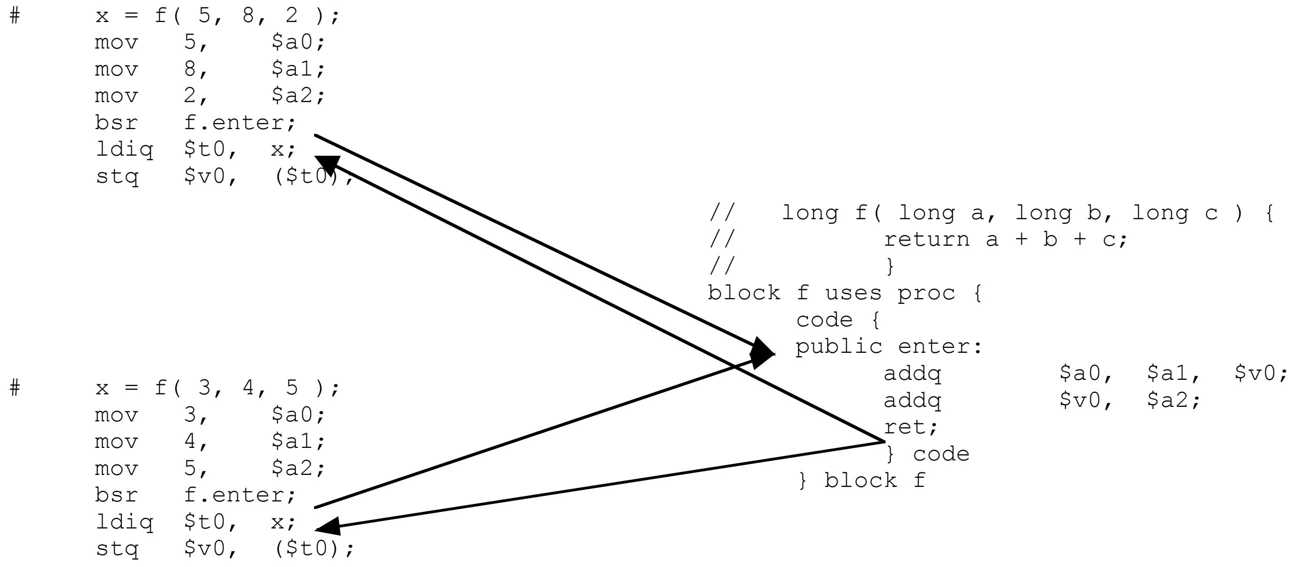
For the declaration of a function:

- Generate code for the body of the function. This code accesses the parameters, modifies local and global variables, and stores the return value in a standard place. On the Alpha, the "saved" registers s0, s1, ... s5 are usually used for local variables, and the return value is stored in register v0.
- Generate an instruction to restore the saved program counter, and return to just after the bsr instruction used to jump to the start of the function. On the Alpha, the ret (return) instruction is used to return to just after the invocation of the function.

For example, if the function f returns the sum of its three parameters, we might write:

```
//   int f( int a, int b, int c ) {
//        return a + b + c;
//        }
block f uses proc {
     code {
     public enter:
          addq      $a0, $a1, $v0;
          addq      $v0, $a2;
          ret;
          } code
     } block f
```

Because a function may be invoked from more than one place, we need to remember where to return to. Thus we need to use the bsr and ret instructions to enter and return from the function, rather than simple branch instructions. Because a function may be invoked with different parameters, we need to evaluate the parameters and store them in the argument registers, rather than accessing them directly.



```
#     x = f( 5, 8, 2 );
      mov   5,    $a0;
      mov   8,    $a1;
      mov   2,    $a2;
      bsr   f.enter;
      ldiq  $t0,  x;
      stq   $v0,  ($t0);
```

```
//   long f( long a, long b, long c ) {
//          return a + b + c;
//          }
block f uses proc {
      code {
      public enter:
            addq        $a0,  $a1,  $v0;
            addq        $v0,  $a2;
            ret;
            } code
      } block f
```

```
#     x = f( 3, 4, 5 );
      mov   3,    $a0;
      mov   4,    $a1;
      mov   5,    $a2;
      bsr   f.enter;
      ldiq  $t0,  x;
      stq   $v0,  ($t0);
```

Example

The function

```
//   int square( int x ) {
//        return x * x;
//        }
block square uses proc {
     code {
     public enter:
          mulq      $a0, $a0, $v0;
          ret;
          } code
     } block square
```

computes the square of its argument.

It can be invoked as

```
//   y = square( 5 );
     ldiq $a0, 5;
     bsr  square.enter;
     ldiq $t0, y;
     stq  $v0, ($t0);
```

§11.2 The special instructions involved in function invocations

The Branch to Subroutine (bsr) instruction

`bsr destAddress;`

The program counter is saved in the ra (return address) register, then the program counter is set to `destAddress`.

There is also a jsr (jump to subroutine) instruction, that can be used when the address of the function has to be computed at run time. We will not use this instruction.

The Return (ret) instruction

`ret;`

The program counter is set to the contents of the ra (return address) register.

§11.3 Function Invocation and Declaration Conventions

When driving, we follow various conventions. For example, in New Zealand, we drive on the left hand side of the road, we obey the traffic lights, when turning we give way to traffic going straight ahead, etc. It would be essentially impossible to drive on the roads, if nobody bothered about which side of the road they drove on. Similarly, it is necessary to follow publicly agreed upon conventions when writing code that has to interface with code generated by compilers or written by other people.

Every computer architecture has standard conventions for invoking and declaring functions. To avoid confusion, the code generated by all compilers must obey these conventions, as must the code generated by assembly language programmers. So long as these conventions are obeyed, the writer of one function does not have to know about the details of the implementation of other functions. It is sufficient to know what they achieve, their public interface, and that they obey the conventions. Most of your program can be written in a high level language, but some functions can be written in assembly language, for efficiency reasons, or because the high level language lacks the expressive power necessary to represent the algorithm. So long as your assembly language obeys the standard conventions, there will be no problem in combining the high level language and assembly language code together.

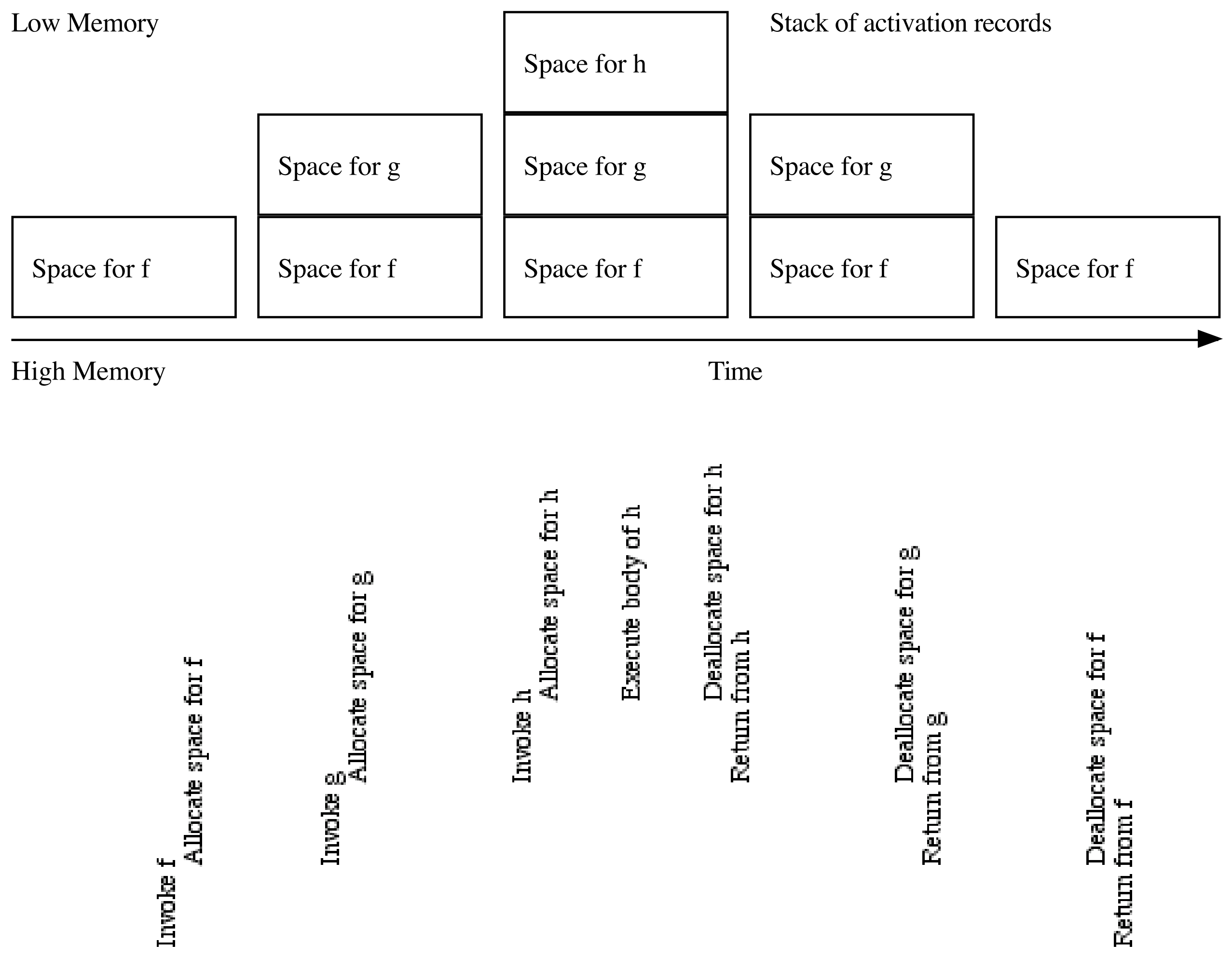
Conventions related to use of the stack

Sometimes functions need local memory in which to store the values of local variables. Arrays need to be stored in memory, because they are too big to fit in a register. Variables passed as reference parameters also need to be stored in memory, because they need to have an address. Local memory is also used for saving the values of registers, that the function wants to use for other purposes.

Most modern computer languages support recursive functions. We can have many instances of the same function that have been invoked but not returned from. Each invocation needs its own local space. This space, called the activation record, call frame, or stack frame for the function, is allocated when the function is invoked, and deallocated when it completes.

Functions are entered and returned from in a stack-like fashion. In other words, if function f invokes function g which invokes function h, then we must return from function h, before we return from function g, and then return from function g, before we return from function f. We can use a stack to allocate the local memory for functions that have been invoked but not returned from.

```
function f( ... ) {
     g( ... );
     }
function g( ... ) {
     h( ... );
     }
```



On the Alpha, the sp (stack pointer) register is used to point to the "top" of the stack (memory space allocated for local space for functions). The stack in fact grows towards low memory. We can allocate space on the stack by subtracting a constant from the stack pointer register, and deallocate space by adding this constant back onto the stack pointer register. The space for local variables can be accessed by using a non-negative displacement from the stack pointer register. Because it is normal to think of the stack growing "up", it is appropriate, when drawing diagrams, to display low addresses at the top of the page, and high addresses at the bottom.

In the Alpha simulator, you can view the stack by scrolling to the bottom (high address end) of the user memory window.

Conventions for the use of registers on the Alpha

Some of the most important conventions related to function invocations involve the specification of how registers should be used, and whether the invoking or the invoked function has responsibility for saving and restoring them.

On the Alpha, the main registers are as follows.

- The program counter.

  This register points to the address of the next instruction to execute. It is always longword aligned.

- 32 integer registers.

0	$v0	This register is used to hold the return value of an integer function. This register may be altered by the invoked function, even if it doesn't return a result (for example, because the invoked function may invoke another function that returns a result).
1-8	$t0-$t7	These are temporary registers used for expression evaluation within a simple statement. They are not normally used to store data between statements. These registers may be altered by the invoked function, so important data can not be left in these registers while another function is invoked.
9-14	$s0-$s5	These are the “saved” registers, usually used to hold the values of local variables (in particular, local variables declared as “register variables” in C). If the invoked function wishes to use these registers, the invoked function must save the registers in its activation record on entry, and restore them on exit. As a consequence, the invoker can behave as if the invoked function never altered the registers.
15	$fp	This register is used to hold the frame pointer (address of the base of the activation record/stack frame/call frame) if needed. In most situations, the address of the activation record is the same as the top of stack, so the sp register can be used as the base address of the activation record, rather than the fp register, and the fp register is never set up. If a function needs to dynamically allocate local space in addition to its activation record (for example, for a dynamically sized local array), it can set the fp register to the base of the activation record, then allocate further space by further decrementing the sp register. The invoked function is responsible for saving and restoring this register.
16-21	$a0-$a5	These registers are used to pass the first six integer type actual parameters. The action of setting up these registers for the invoked function overwrites the values of the parameters for the invoker. Hence a function that invokes another function must save the values of these registers on entry, either in its activation record, or in saved registers.
22-25	$t8-$t11	These are additional temporary registers used for expression evaluations.
26	ra	This register is used to hold the return address of a function. The program counter is saved in this register by the bsr instruction and restored from this register by the ret instruction. Hence a function that invokes another function must save the value of this register in its activation record on entry, and restore it before exit.
27	$pv	This register is used by the jsr pseudoinstruction to hold the entry point of the current function. The jsr pseudoinstruction effectively loads the destination address into this register, then does an indirect jump via this register. We will not make use of this register.

Function invocations and declarations

28	$at	This register is used by the UNIX assembler to implement pseudoinstructions, such as instructions for which the literal operands are outside the range permitted by the real instruction. It should not be used directly by assembly language programmers. My assembler does not actually make use of this register.
29	$gp	This register is used to hold the global pointer. The global pointer points to a table containing the values of constants, such as the addresses of functions and global variables. This table is needed because the number of bits used to represent a constant in an instruction are too few to represent a 64 bit value. The ldiq pseudoinstruction is converted into a ldq instruction. The constant is stored in the global table, and accessed as an offset from the global pointer. The global pointer register is set up by the operating system when the program is loaded, and stays constant throughout the execution of the program.
30	$sp	This register is used to hold the stack pointer (the address of the “top” of stack). The invoked function allocates space for itself on the stack by subtracting the size of the activation record from the stack pointer. On return, the invoked function deallocates the stack space by adding the size of the activation record to the stack pointer.
31	$zero	Always has the value 0.

- 32 floating point registers.

  There are similar conventions for the use of floating point registers. Registers are divided up into registers for arguments, temporary registers, saved registers, and two registers (to allow for complex numbers) to store the return value.

Conventions for invoking functions on the Alpha

- Evaluate the parameters. The first six parameters are stored in registers $a0, $a1, $a2, ... $a5. (Any additional parameters are stored at the low address end of the activation record of the invoker. However, we will never deal with functions with more than six parameters.)
- A bsr instruction is used to invoke the function. The program counter is saved in the $ra (return address) register, and the program counter is set to the address of the start of the function.
- The invoker can assume that, after the invocation
  - The result of the function will be in register $v0.
  - The “saved” registers will contain the same values they had before the invocation.
  - The stack pointer register will be the same as it was before the invocation.

```
mov  ..., $a0;          //  Set up $a0
mov  ..., $a1;          //  Set up $a1
mov  ..., $a2;          //  Set up $a2
...                     //  ...
bsr  f.enter;           //  Invoke f
mov  $v0, ...;          //  Assign return value
```

Conventions for declaring functions on the Alpha

- If the function needs any local memory, allocate space for the activation record on the stack by subtracting the number of bytes needed from the stack pointer. This is usually done by the instruction “lda $sp, -frameSize($sp);”.
- If the function invokes another function, save the return address register in the activation record of the function. (This is because invoking another function will overwrite the return address register.)
- If the function allocates space for local arrays, save the frame pointer register in the activation record of the function.
- If the function wants to make use of the “saved” registers for its own local variables, or saving the arguments, save these registers in the activation record of the function.
- If the function invokes another function, save the argument registers in “saved” registers. (This is better than saving them directly in the activation record, because heavy use is normally made of the arguments, and it is faster to access them via registers than from memory.)
- Temporary registers can be used, without needing to save and restore them.
- Evaluate the body of the function. Use the saved registers for simple local variables that can fit into registers, and are not referred to by “reference”. Use memory on the stack for local arrays, etc.
- Store the return value in register $v0.
- Restore any registers that were saved on entry to the function.
- If the function allocated any local memory for an activation record, deallocate this space by adding the size of the space to the stack pointer. This is usually done by the instruction “lda $sp, +frameSize($sp)”.
- A ret (return) instruction is used to return to just after the bsr instruction used to invoke the function.

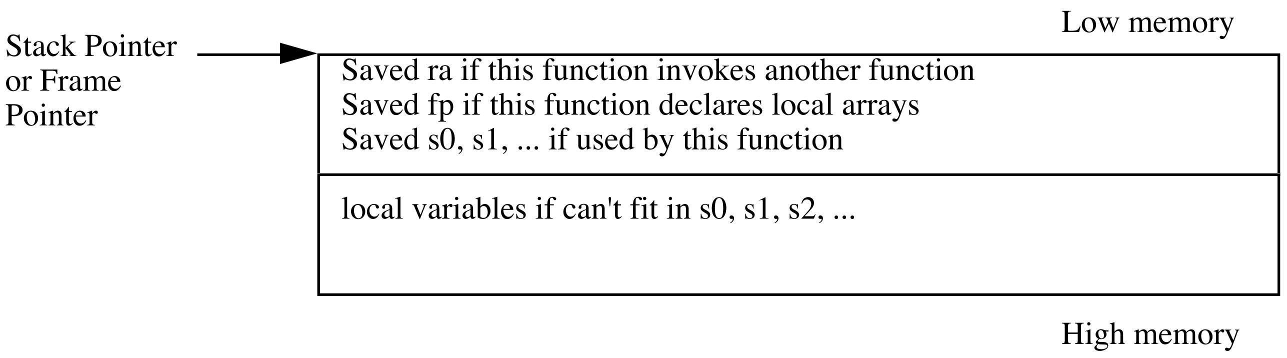
For convenience, we define a block, proc, with a local section with symbolic names for the offsets for the saved values of the $ra, $fp, $s0, $s1, $s2, $s3, $s4, and $s5 registers. Because a register contains 8 bytes, the offsets saveRet, savFP, sav0, sav1, sav2, ... are 0, 8, 16, 24, 32, ...

```
block proc {
     local {
     protected savRet:  quad;
     protected savFP:   quad;
     protected sav0:    quad;
     protected sav1:    quad;
     protected sav2:    quad;
     protected sav3:    quad;
     protected sav4:    quad;
     protected sav5:    quad;
     protected sav6:    quad;
          } local
     } block proc

block f uses proc {
     code {
     public enter:
```

```
//--------------------------------------------------------
                                    //   Entry Code
//--------------------------------------------------------
         lda  $sp, -frameSize($sp);   //   Allocate space on stack
         stq  $ra, savRet($sp);       //   Save $ra on stack
         stq  $s0, sav0($sp);         //   Save $s0 on stack
         stq  $s1, sav1($sp);         //   Save $s1 on stack
         ...                          //   ...
//--------------------------------------------------------
                                    //   Initialisation of variables
//--------------------------------------------------------
     init:
         mov  $a0, $s0;               //   Save $a0 in $s0
         mov  $a1, $s1;               //   Save $a1 in $s1
                                      //   (only needed if invokes
                                      //   another function)
         ...
//--------------------------------------------------------
                                    //   Body of function
//--------------------------------------------------------
     body:
         ...
         mov  ..., $v0;               //   Store result in $v0
//--------------------------------------------------------
                                    //   Exit Code
//--------------------------------------------------------
     return:
         ...
         ldq  $s1, sav1($sp);         //   Restore $s1
         ldq  $s0, sav0($sp);         //   Restore $s0
         ldq  $ra, savRet($sp);       //   Restore $ra
         lda  $sp, +frameSize($sp);   //   Deallocate space on stack
         ret;
         } code
     } block f
```

The Layout for a activation record


Low memory
Stack Pointer or Frame Pointer
Saved ra if this function invokes another function
Saved fp if this function declares local arrays
Saved s0, s1, ... if used by this function
local variables if can't fit in s0, s1, s2, ...
High memory

Some portions of the activation record may be omitted, and in fact for simple functions only the portion used to save registers is likely to exist. Even this may be omitted for “leaf” functions that do not invoke other functions, and do not use the saved registers.

The activation record must be padded to a multiple of 8 bytes, so that all data is quadword aligned.

§11.4 Reference parameters and Pointers

In C, we can pass the address of a simple variable, array, or record to a function. These are often called reference parameters. We can also have reference variables, that point to an address of another variable. In Java we have something equivalent. Variables of type corresponding to an array or class are really pointers to objects in memory.

By accessing the variable indirectly through the reference, we can modify its value.

For example, suppose we want to write a function that takes the address of two integer variables as parameters, and swaps their values. We could write:

```
void swap( int *a, int *b ) {
     register int temp;
     temp = *a;
     *a = *b;
     *b = temp;
     }
```

Note: *a means (the contents of) the address pointed to by a. The parameters a and b are not altered, only the data at the addresses pointed to by them.

In assembly language, this is:

```
block swap uses proc {
     abs {
          a    =    a0;
          b    =    a1;
          } abs
     code {
     public enter:
          ldq       $t0,      ($a);
          ldq       $t1,      ($b);
          stq       $t1,      ($a);
          stq       $t0,      ($b);
          ret
          } code
     } block swap
```

We can invoke the function swap(), with the addresses of variables as parameters. In C:

```
int x = 3, y = 4;
void main() {
     swap( &x, &y );
     }
```

Note: &x means the address of x.

In assembly language, this is:

```
block main uses proc {
     data {
     x:   quad 3;
     y:   quad 4;
          } data
     code {
     public enter:
          ldiq      $a0,      x;        //   a0 = &x;
          ldiq      $a1,      y;        //   a1 = &y;
          bsr       swap.enter;         //   swap( &x, &y );
          clr       $a0;                //   exit( 0 );
          bsr       exit.enter;
          } code
     } block main
```

Suppose we want to make x and y local to a function, then invoke swap. In C:

```
void f() {
     int x = 3, y = 4;
     swap( &x, &y );
     }
```

Function invocations and declarations

We cannot store x and y in registers, because only memory can have an address. Hence we have to store them in the activation record. In assembly language, this is:

```
block f extends proc.sav0 uses proc {
     local {
     x:   quad;
     y:   quad;
     size:
          } local
     code {
     public enter:
          lda      $sp,      -size($sp);   //   Allocate space
          stq      $ra,      savRet($sp);  //   Save ra
     init:
          mov      3,        $t0;          //   int x = 3;
          stq      $t0,      x($sp);
          mov      4,        $t0;          //   int y = 4;
          stq      $t0,      y($sp);
     body:
          lda      $a0,      x($sp);       //   swap( &x, &y );
          lda      $a1,      y($sp);
          bsr      swap.enter;
     return:
          ldq      $ra,      savRet($sp);  //   Restore ra
          lda      $sp,      size($sp);    //   Deallocate space
          ret
          } code
     }block f
```

A local section is used to define identifiers corresponding to offsets within an activation record. It does not actually allocate static memory. The extends option specifies the initial offset for labels in the local section of the block. Space within the activation record can be allocated by memory allocation statements, such as quad.

We could write a function that sums the elements of an array, whose address is passed as a parameter.

```
//   int sum( int[] array, int max ) {
//        int total = 0;
//        int i;
//        for ( i = 0; i < max; i++ )
//             total += array[ i ];
//        return total;
//        }

block sum uses proc {
     abs {
          array     =    s0;
          max       =    s1;
          i         =    s2;
          total     =    s3;
          } abs
     code {
     public enter:
          lda      $sp, -sav4($sp);
          stq      $ra, savRet($sp);
          stq      $s0, sav0($sp);
          stq      $s1, sav1($sp);
          stq      $s2, sav2($sp);
          stq      $s3, sav3($sp);
     init:
          mov      $a0, $array;
          mov      $a1, $max;
     body:
          clr       $total;                      //   total = 0;
          {
          for:                              //   for ( i = 0; i < max; i++ )
               clr       $i;
          while:
               cmplt     $i,       $max,     $t0;
               blbc $t0, end;
          do:
               s8addq    $i,       $array,   $t0;//        total += array[ i ];
               ldq       $t0, ($t0);
               addq $total,   $t0;
          continue:
               addq $i,  1;
               br        while;
          end:
          }
          mov       $total,   $v0;
     return:
          ldq       $s3, sav3($sp);
          ldq       $s2, sav2($sp);
          ldq       $s1, sav1($sp);
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav4($sp);
          ret;
          } code
     } block sum
```

We can invoke the function sum(), with the address of an array, and the size of the array as parameters.

```
//   int BUFFERSIZE = 20;
//   int DATASIZE = 10;
//   char buffer[ BUFFERSIZE + 1 ];
//   int array[ DATASIZE ];
//   void main() {
//        int maxArray;
//        int result;
//        for ( maxArray = 0; maxArray < DATASIZE; maxArray++ ) {
//             print( "Enter a number (or return to finish): " );
//             readLine( buffer, BUFFERSIZE );
//             if ( buffer[ 0 ] == 0 )
//                  break;
//             array[ maxArray ] = Number.fromString( buffer, 10 );
//             }
//        print( "Sum of elements of the array:\n" );
//        printArray( array, maxArray );
//        result = sum( array, maxArray );
//        printf( " is %d\n", result );
//        newline();
//        exit( 0 );
//        }
block main uses proc {
     abs {
          BUFFERSIZE     =    20;
          DATASIZE  =    10;
          maxArray  =    s0;
          result         =    s1;
          i              =    s2;
          j              =    s3;
          } abs
```

Function invocations and declarations

```
     const {
               align;
          message1:
               asciiz   "Enter a number (or return to finish): ";
               align;
          message2:
               asciiz   "Sum of elements of the array:\n";
               align;
          format:
               asciiz " is %d\n";
          } const
     data {
          align;
     buffer:
          byte [ BUFFERSIZE + 1 ];
     array:
          quad [ DATASIZE ];
          } data
     code {
     public enter:
          {
          for:                    //   for ( maxArray = 0;
                                  //        maxArray < DATASIZE; maxArray++ ) {
               clr      $maxArray;
          while:
               cmplt    $maxArray,    DATASIZE, $t0;
               blbc $t0, end;
          do:
               ldiq $a0, message1; //        print(
                                  //        "Enter a number (or return to finish):
" );
               bsr      IO.print.enter;
               ldiq $a0, buffer;  //        readLine( buffer, BUFFERSIZE );
               ldiq $a1, BUFFERSIZE;
               bsr      IO.readLine.enter;
               ldiq $t0, buffer;  //        if ( buffer[ 0 ] == 0 )
               ldbu $t0, ($t0);
               beq      $t0, end; //             break;
               ldiq $a0, buffer;  //        array[ maxArray ] = fromString(
buffer, 10 );
               ldiq $a1, 10;
               bsr      Number.fromString.enter;
               ldiq $t0, array;
               s8addq   $maxArray,    $t0, $t0;
               stq      $v0, ($t0);
          continue:
               addq $maxArray,    1;
               br       while;    //        }
          end:
          }
          ldiq $a0, message2;     //   print( "Sum of elements of the array:\n" );
          bsr      IO.print.enter;//  printArray( array, maxArray );
          ldiq $a0, array;
          mov      $maxArray,    $a1;
          bsr      printArray.enter;
          ldiq $a0, array;        //   result = sum( array, maxArray );
          mov      $maxArray,    $a1;
          bsr      sum.enter;
          mov      $v0, $result;
          ldiq $a0, format;
          mov      $result,  $a1; //   printf( " is %d\n", result );
          bsr      IO.printf.enter;
          clr      $a0;           //   exit( 0 );
          bsr      Sys.exit.enter;
          } code
     } block main
```

Exercise PRINTARRAY_ERROR

The following program is meant to print out the index and value of the elements of the array, and should generate the output:

```
   0       3
   1      14
   2      15
   3      92
```

However, it has at least 10 errors in it. Indicate and correct at least 10 errors.

```
entry main.enter;

import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/string.lib.s";
import "../IMPORT/number.lib.s";
import "../IMPORT/io.lib.s";

data {
     array:
          quad 3;
          quad 14;
          quad 15;
          quad 92;
     } data

//   void printArray( int[] array, int max ) {
//        int i;
//        for ( i = 0; i < max; i++ )
//             printf( "%4d%8d\n", i, array[ i ] );
//        }

block printArray uses proc {
     abs {
          array     =    s0;
          max  =    s1;
          i    =    s2;
          } abs
     const {
               align;
          format:
               asciiz "%4d%8d\n";
          } const
     code {
     public enter:
          lda  $sp, -sav3($sp);
          stq  $ra, savRet($sp);
          stq  $s0, sav0($sp);
          stq  $s1, sav1($sp);
          stq  $s2, sav2($sp);
     init:
          mov  $a0, $array;
     body:

          {
```

```
          for:
               clr  $i;
          while:
               cmplt     $i,  max, $t0;
               blbc end;
          do:
               ldq  $a0, format;
               ldiq $a1, i;
               addq $i,  $array,   $t0;
               ldq  $a3, $t0;
               bsr  IO.printf.enter;
          continue:
               addq $i,  1;
               br   end;
          end:
          }
     return:
          ldq  $s2, sav2($sp);
          ldq  $s1, sav1($sp);
          ldq  $s0, sav0($sp);
          ldq  $ra, savRet($sp);
          lda  $sp, +sav2($sp);

          } code
     } block printArray
//   int main() {
//        printArray( array, 4 );
//        }
block main uses proc {
     code {
     public enter:
          ldiq $a0, array;
          ldq  $a1, 4;
          bsr  printArray;

          } code
     } block main
```

§11.5 Some programming exercises to try

Strings

Write a function void toLower(char *s) that

- Converts the characters in a string s to lower case.

Write a function void substring(char *dest, char *source, int start, int finish) that

- Copies the substring of source, from index start, to just before index finish, and stores in dest.

Write a function int countChars(char *source, char low, char high) that

- Counts the characters c in source that satisfy low <= c <= high.

Write a function void extractChars(char *dest, char *source, char low, char high) that

- Copies the characters c in source that satisfy low <= c <= high into the memory starting at the address dest.

Write a function int compareIgnoreCase(char *s, char *t) that

- Compares two strings s and t, as if they had been converted to lower case, and returns an integer indicating whether s < t, s == t s > t.

Function invocations and declarations

- Does not modify the strings themselves.
- Makes use of functions in string.lib.s.

Write a function int findCharIndex(char c, char *s) that

- Find the index of the first occurrence of char c in string s.
- Returns the index, or -1, if the char is not found.

Write a function int findLastCharIndex(char c, char *s) that

- Find the index of the last occurrence of char c in string s.
- Returns the index, or -1, if the char is not found.

Write a function int startsWith(char *prefix, char *s) that

- Returns true if the string s starts with the string prefix.

Write a function int indexOf(char *s, char *t) that

- Returns the index of the first occurrence of the string t as a substring in the string s, or -1 if it does not occur.

Write a function void printTrim(char *s, char padChar, int size) that

- Prints the first size characters of s, padding with padChar if s has less than size characters.

Write a function void copyTrim(char *dest, char *source, char padChar, int size) that

- Copies the string starting at the address source to the memory starting at the address dest.
- Copies at most size bytes.
- Does not copy beyond the null byte terminator.
- Pads dest with the specified padChar, if source has less than size chars.

Write a function int compareTrim(char *s, char *t, int size) that

- Compares two strings s and t, and returns an integer indicating whether s < t, s == t s > t.
- Only compares the first size characters.

Write a function void shiftUp(char c, char *s) that

- Shifts the text in s up (right) by one byte, deleting the last character, and inserting character c at the start.

Write a function void shiftDown(char c, char *s) that

- Shifts the text in s down (left) by one byte, deleting the first character, and inserting character c at the end.

Write a function void shift(char c, char *s, int count) that

- Shifts the text in s down (left) by -count bytes or up (right) by +count bytes, deleting any characters that do not fit inside the original string, and filling the space with the character c.

Write a function char *findCharPosition(char c, char *s) that

- Assumes the characters in string s are sorted in order.
- Returns the position of the first char in string s that is >= c.
- Returns the address just beyond the end of the string, if all chars are < c.

Function invocations and declarations

Write a function void insertChar(char c, char *s) that

- Assumes the characters in string s are sorted in order.
- If the char c does not already exist in s, inserts it in the appropriate place, to keep the text sorted.

Write a function void deleteChar(char c, char *s) that

- If the char c exists in s, deletes the first occurrence.

Write a function void deleteAllChar(char c, char *s) that

- Deletes all occurrences of c in s.

Write a function void replaceAllChar(char *s, char from, char to) that

- Replaces all occurrences of char from in s by char to.

Write a function void translate(char *s, char *from, char *to) that

- Replaces all occurrences of chars in from in s by the char in the corresponding position in to.

Write a function void copyOnly(char *dest, char *source, char *onlyChars) that

- Copies the chars in source to dest, that exist in onlyChars.

Write a function void copyExcept(char *dest, char *source, char *exceptChars) that

- Copies the chars in source to dest, that do not exist in exceptChars.

Write a function void copyMerge(char *dest, char *source1, char *source2) that

- Assumes the characters in source1 and source2 are sorted.
- Copies the characters in source1 and source2 into dest.
- Maintains the order.
- Deletes duplicates.

Memory

Write a function void fillMem(char *addr, char fillChar, int size) that

- Sets size bytes of memory starting at address addr to the specified fillChar.

Write a function void copyMem(char *dest, char *source, int size) that

- Copies the contents of size bytes of memory starting at the address source to the memory starting at the address dest.
- Takes into account that the memory may overlap.

Write a function void copyMem(char *dest, char *source, int size) that

- Copies the contents of size bytes of memory starting at the address source to the memory starting at the address dest.
- Assumes the memory is at an address divisible by 8, and that size is divisible by 8, and uses ldq and stq, rather than ldbu and stb.

Integer Arrays

Write a function int sum(int array[], int size) that

- Returns the sum of the elements of the array, which contains size elements.

Write a function int max(int array[], int size) that

- Returns the maximum of the elements of the array, which contains size elements.
- Takes into account that the elements may be negative.
- Takes into account that the size may be 0.

Write a function void add(int dest[], int src1[], int src2[], int size) that

- Computes the sum of the arrays src1 and src2 and puts the result in the array dest (performing vector addition). All arrays have the specified size elements.

Write a function int dotProduct(int src1[], int src2[], int size) that

- Returns the dot product of the arrays src1 and src2.

Design a representation for one and two dimensional arrays that includes the size of the array as part of the data structure.

- Rewrite the above functions to use this information, rather than passing the size as a parameter.
- Write functions to perform matrix arithmetic.

Sorting and Searching

Write a function int insert(char *array[], char *s) that

- Assumes the array contains a sorted list of pointers to strings, and that the end of the list is indicated by a null address.
- Searches for the position of the string s in the array.
- If it does not find it, inserts it in the array, and shuffles the following elements up to make space.
- Makes sure that the array is still terminated by a null address.
- Returns the index of the string in the array.

Write a function void sort(char *array[]) that

- Assumes the array contains a list of pointers to strings, and that the end of the list is indicated by a null address.
- Sorts the elements into order, using a variety of different sorting algorithms.

Write a function int binarySearch(int array[], int value, int low, int high) that

- Assumes that array is a sorted array of integers.
- Performs a binary search for the value in array, from indexes low to high.
- Returns the index of the element, or -1, if it is not found.
- Uses recursion.

Input/Output, and conversion of text to a number

Write a function int toNumber(char *s) that

Function invocations and declarations

- Processes text in s starting with an optional '+' or '-', followed by decimal digits, and converts the decimal number into internal form.
- Returns the number as its result.

Write a function int toNumber(char **sVar) that

- Does the same as the above, but takes the address of a string variable, rather than the address of a string.
- Updates the variable to point to the text just after the decimal number.

Write a function int readNum() that

- Reads characters until it reads in a decimal digit or '+' or '-'.
- Reads characters until it gets a non-digit.
- Processes the optional sign and digits to convert a decimal number into internal form.
- Returns the number as its result.

§11.6 Recursion

When writing recursive functions, we have to be particularly careful about saving and restoring registers. There will always be conflicts between the registers used by the invoking and the invoked function (they are after all the same function). Consider the following C program, that generates Pascal's triangle rather inefficiently.

```
#define MAX 3
int comb( int n, int r ) {
     if ( r == 0 || r == n )
          return 1;
     else
          return comb( n - 1, r - 1 ) + comb( n - 1, r );
     }

int main( int argc, char *argv[], char *arge[] ) {
     register int n, r;
     for ( n = 0; n <= MAX; n++ ) {
          for ( r = n; r < MAX; r++ )
               printf( "    " );
          for ( r = 0; r <= n; r++ )
               printf( "%8d", comb( n, r ) );
          printf( "\n" );
          }
     }
```

This generates output

```
               1
          1         1
     1         2         1
1         3         3         1
```

and invokes the functions as shown below.

```
Enter comb( 0, 0 )
Exit comb( 0, 0 )

Enter comb( 1, 0 )
Exit comb( 1, 0 )
Enter comb( 1, 1 )
Exit comb( 1, 1 )

Enter comb( 2, 0 )
```

```
Exit comb( 2, 0 )
Enter comb( 2, 1 )
     Enter comb( 1, 0 )
     Exit comb( 1, 0 )
     Enter comb( 1, 1 )
     Exit comb( 1, 1 )
Exit comb( 2, 1 )
Enter comb( 2, 2 )
Exit comb( 2, 2 )

Enter comb( 3, 0 )
Exit comb( 3, 0 )
Enter comb( 3, 1 )
     Enter comb( 2, 0 )
     Exit comb( 2, 0 )
     Enter comb( 2, 1 )
          Enter comb( 1, 0 )
          Exit comb( 1, 0 )
          Enter comb( 1, 1 )
          Exit comb( 1, 1 )
     Exit comb( 2, 1 )
Exit comb( 3, 1 )
Enter comb( 3, 2 )
     Enter comb( 2, 1 )
          Enter comb( 1, 0 )
          Exit comb( 1, 0 )
          Enter comb( 1, 1 )
          Exit comb( 1, 1 )
     Exit comb( 2, 1 )
     Enter comb( 2, 2 )
     Exit comb( 2, 2 )
Exit comb( 3, 2 )
Enter comb( 3, 3 )
Exit comb( 3, 3 )
```

We can write comb in assembly language

```
//   int comb( int n, int r ) {
//        if ( r == 0 || r == n )
//             return 1;
//        else
//             return comb( n - 1, r - 1 ) + comb( n - 1, r );
//        }

block comb uses proc {
     abs {
          n         =    s0;
          r         =    s1;
          temp      =    s2;
          } abs
     code {
     public enter:
          lda       $sp, -sav3($sp);
          stq       $ra, savRet($sp);
          stq       $s0, sav0($sp);
          stq       $s1, sav1($sp);
          stq       $s2, sav2($sp);
     init:
          mov       $a0, $n;
          mov       $a1, $r;
     body:
          {
          if:
               beq       $r,  then;
               cmpeq     $r,  $n,  $t0;
               blbc      $t0, else;
          then:
               mov       1,   $v0;
               br        end;
          else:
               subq      $n,  1,   $a0;
               subq      $r,  1,   $a1;
               bsr       comb.enter;
               mov       $v0, $temp;
               subq      $n,  1,   $a0;
               mov       $r,  $a1;
               bsr       comb.enter;
               addq      $v0, $temp;
          end:
          }
     return:
          ldq       $s2, sav2($sp);
          ldq       $s1, sav1($sp);
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav3($sp);
          ret;
          } code
     } block comb
```

The activation record for comb is made up of the saved values of ra, s0, s1, s2. Note that I had to move the result of the invocation of "comb(n - 1, r - 1)" to a saved register, so that it would not get overwritten by the invocation of "comb(n - 1, r)". The saved register ends up being saved on the stack by the entry code for the recursive invocation of the function.

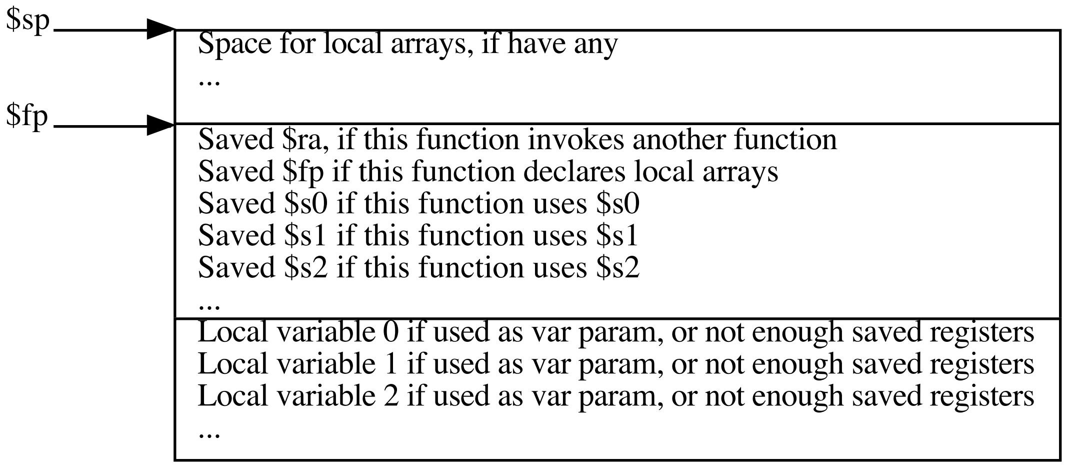
§11.7 Local Arrays

We can write functions that declare local arrays. We need to allocate space for the local array on the stack. Because the stack pointer no longer points to the base of the activation record, we need another register, the fp (frame pointer) register to point to the activation record. The template for the assembly language for such functions something like the following:

```
block f uses proc {
     abs {
          arrayPtr =    ...;           //   A saved register
          } abs
     code {
     public enter:
//--------------------------------------------------------------
//   Entry Code
//--------------------------------------------------------------
          lda       $sp,      -frameSize($sp);
          stq       $ra,      savRet($sp);
          stq       $fp,      savFP($sp);
          stq       $s0,      sav0($sp);
          stq       $s1,      sav1($sp);
          ...
          mov       $sp,      $fp       //   Set frame pointer
     init:
```

```
//--------------------------------------------------------------
//   Allocate space for the array
//--------------------------------------------------------------
          // Allocate array
          lda      $sp,      -elementSize*arraySize($sp);
          mov      $sp,      $arrayPtr;
//--------------------------------------------------------------
//   Body of function
//--------------------------------------------------------------
     body:
     ...
//--------------------------------------------------------------
//   Exit Code
//--------------------------------------------------------------
     return:
          //   Deallocate array
          mov      $fp,      $sp;
          ...
          ldq      $s1,      sav1($sp);
          ldq      $s0,      sav0($sp);
          ldq      $fp,      savFP($sp);
          ldq      $ra,      savRet($sp);
          lda      $sp,      +frameSize($sp);
          ret;
          } code
     } block f
```

If the array has a fixed size, it is possible to allocate space for the array within the activation record. However, the above system allows us to allocate arrays with a size that depends on the parameters passed to the function. If we have local variables in the activation record, they can be accessed by offsets from the frame pointer.


$sp
$fp
Space for local arrays, if have any
...
Saved $ra, if this function invokes another function
Saved $fp if this function declares local arrays
Saved $s0 if this function uses $s0
Saved $s1 if this function uses $s1
Saved $s2 if this function uses $s2
...
Local variable 0 if used as var param, or not enough saved registers
Local variable 1 if used as var param, or not enough saved registers
Local variable 2 if used as var param, or not enough saved registers
...

For example, the following C program generates Pascal’s triangle a row at a time, storing the result in an array.

```
int MAX = 3;

void printRow( int data[], int n ) {
     for ( int i = n; i < MAX; i++ )
          print( "    " );
     for ( int i = 0; i <= n; i++ )
          printf( “%8d”, data[ i ] );
     newline();
     }

void genRow( int row[], int n ) {
     int prevRow[ n ];
     int r;
     row[ 0 ] = row[ n ] = 1;
     if ( n > 0 ) {
          genRow( prevRow, n - 1 );
          for ( r = 1; r < n; r++ )
               row[ r ] = prevRow[ r - 1 ] + prevRow[ r ];
          }
     printRow( row, n );
     }


int main( int argc, char *argv[], char *arge[] ) {
     int row[ MAX + 1 ];
     genRow( row, MAX );
     exit( 0 );
     }
```

Translating the above into Alpha assembly language, we get the following.

```
//   int MAX = 3;

abs {
     MAX  =    3;
     } abs

//   void printRow( int array[], int n ) {
//        int i;
//        for ( i = n; i < MAX; i++ )
//             print( "    " );
//        for ( i = 0; i <= n; i++ )
//             printf( "%8d", array[ i ] );
//        newline();
//        }
```

Function invocations and declarations

```
block printRow uses proc {
     abs {
          array     =    s0;
          n         =    s1;
          i         =    s2;
          } abs
     const {
          space4:
               asciiz    "    ";
          format:
               asciiz "%8d";
          } const
     code {
     public enter:
          lda       $sp, -sav3($sp);
          stq       $ra, savRet($sp);
          stq       $s0, sav0($sp);
          stq       $s1, sav1($sp);
          stq       $s2, sav2($sp);
     init:
          mov       $a0, $array;
          mov       $a1, $n;
     body:
          {
          for:                         //        for ( i = n; i < MAX; i++ )
               mov       $n,       $i;
          while:
               cmplt     $i,       MAX, $t0;
               blbc $t0, end;
          do:
               ldiq $a0, space4;       //             print( "    " );
               bsr       IO.print.enter;
          continue:
               addq $i,       1;
               br        while;
          end:
          }
          {
          for:                         //        for ( i = 0; i < n; i++ )
               clr       $i;
          while:
               cmple     $i,       $n,  $t0;
               blbc $t0, end;
          do:
               ldiq $a0, format;       //             printf( "%8d", array[ i ] );
               s8addq    $i,       $array,   $t0;
               ldq       $a1, ($t0);
               bsr       IO.printf.enter;
          continue:
               addq $i,       1;
               br        while;
          end:
          }
          bsr       IO.newline.enter;  //        newline();
     return:
```

Function invocations and declarations

```
          ldq       $s2, sav2($sp);
          ldq       $s1, sav1($sp);
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav3($sp);
          ret;
          } code
     } block printRow

//   void genRow( int row[], int n ) {
//       int prevRow[ n ];
//        //   Actually not legal to have dynamic size allocation in C
//       int r;
//       row[ 0 ] = row[ n ] = 1;
//       if ( n > 0 ) {
//           genRow( prevRow, n - 1 );
//           for ( r = 1; r < n; r++ )
//               row[ r ] = prevRow[ r - 1 ] + prevRow[ r ];
//           }
//       printRow( row, n );
//       }
//
block genRow uses proc {
     abs {
          row            =    s0;
          n              =    s1;
          prevRow        =    s2;
          r              =    s3;
          } abs
     code {
     public enter:
          lda       $sp, -sav4($sp);
          stq       $ra, savRet($sp);
          stq       $fp, savFP($sp);
          stq       $s0, sav0($sp);
          stq       $s1, sav1($sp);
          stq       $s2, sav2($sp);
          stq       $s3, sav3($sp);
          mov       $sp, $fp;
     init:
          mov       $a0, $row;
          mov       $a1, $n;
          sll       $n,       3,        $t0; //   int prevRow[ n ];
          subq $sp, $t0;
          mov       $sp, $prevRow;
```

```
body:
     mov       1,        $t0;           //   row[ 0 ] = row[ n ] = 1;
     stq       $t0, ($row);
     s8addq    $n,  $row,     $t1;
     stq       $t0, ($t1);
     {
     if:                                //   if ( n > 0 ) {
          ble       $n,       end;
     then:                              //        genRow( prevRow, n - 1 );
          mov       $prevRow, $a0;
          subq $n,       1,   $a1;
          bsr       genRow.enter;
          {
          for:                          //        for ( r = 1; r < n; r++ )
               mov       1,        $r;
          while:
               cmplt     $r,  $n,  $t0;
               blbc $t0, end;
          do:                           //             row[ r ] =
                                        //                  prevRow[ r - 1 ]
                                        //                  + prevRow[ r ];
               s8addq    $r,       $row,     $t0;
               subq $r,       1,        $t1;
               s8addq    $t1, $prevRow, $t1;
               ldq       $t1, ($t1);
               s8addq    $r,       $prevRow, $t2;
               ldq       $t2, ($t2);
               addq $t1, $t2;
               stq       $t1, ($t0);
          continue:
               addq $r,  1;
               br        while;
          end:
          }
     end:                               //        }
     }
     mov       $row,     $a0;           //   printRow( row, n );
     mov       $n,       $a1;
     bsr       printRow.enter;
return:                                 //   }
     mov       $fp, $sp;
     ldq       $s3, sav3($sp);
     ldq       $s2, sav2($sp);
     ldq       $s1, sav1($sp);
     ldq       $s0, sav0($sp);
     ldq       $fp, savFP($sp);
     ldq       $ra, savRet($sp);
     lda       $sp, +sav4($sp);
     ret;
     } code
} block genRow
```

```
//   int main( int argc, char *argv[], char *arge[] ) {
//       int row[ MAX + 1 ];
//       genRow( row, MAX );
//        exit( 0 );
//       }
//

block main uses proc {
     code {
     public enter:
          subq $sp, 8*(MAX+1);          //   int row[ MAX + 1 ];
          mov      $sp, $a0;            //   genRow( row, MAX );
          mov      MAX, $a1;
          bsr      genRow.enter;
          clr      $a0;                 //   exit( 0 );
          bsr      Sys.exit.enter;
          } code
     } block main
```

Note that it is not possible to return a local array as the result of a function, because the space will be deallocated on return from the function, and overwritten by the next function invocation.

What does the stack look like, when at the maximum level of recursion? All the arrays for each row of Pascal's triangle have been set up, and the 1's at each end have been assigned, but the middle values have not been filled in.

General Registers

Program Counter

```
    pc       .genRow.end     At .genRow.end in genRow( row, 0 )
```

Integer Registers

```
    s0          1ffff20     row in genRow( row, 0 )
    s1                0     n in genRow( row, 0 )
    s2          1fffef0     prevRow in genRow( row, 0 ) (array of size 0)
    s3                0     r in genRow( row, 0 )
    fp          1fffef0     frame pointer for genRow( row, 0 )
    sp          1fffef0     top of stack
```

Stack Memory

prevRow in genRow(row, 0) (size 0)

activation record for genRow(row, 0)

```
01fffef0  .genRow...for     saved ra  to genRow( row, 1 )
01fffef8        1ffff28     saved fp for genRow( row, 1 )
01ffff00        1ffff58     saved s0  row in genRow( row, 1 )
01ffff08              1     saved s1  n in genRow( row, 1 )
01ffff10        1ffff20     saved s2  prevRow in genRow( row, 1 )
01ffff18              0     saved s3  r in genRow( row, 1 )
```

prevRow in genRow(row, 1)

```
01ffff20              1     prevRow[ 0 ]
```

activation record for genRow(row, 1)

```
01ffff28  .genRow...for     saved ra  to genRow( row, 2 )
01ffff30        1ffff68     saved fp for genRow( row, 2 )
01ffff38        1ffff98     saved s0  row in genRow( row, 2 )
01ffff40              2     saved s1  n in genRow( row, 2 )
01ffff48        1ffff58     saved s2  prevRow in genRow( row, 2 )
01ffff50              0     saved s3  r in genRow( row, 2 )
```

prevRow in genRow(row, 2)

```
01ffff58              1     prevRow[ 0 ]
```

```
01ffff60              1     prevRow[ 1 ]
```

activation record for genRow(row, 2)

```
01ffff68  .genRow...for     saved ra  to genRow( row, 3 )
01ffff70        1ffffb0     saved fp for genRow( row, 3 )
01ffff78        1ffffe0     saved s0  row in genRow( row, 3 )
01ffff80              3     saved s1  n in genRow( row, 3 )
01ffff88        1ffff98     saved s2  prevRow in genRow( row, 3 )
01ffff90              0     saved s3  r in genRow( row, 3 )
```

prevRow in genRow(row, 3)

```
01ffff98              1     prevRow[ 0 ]
01ffffa0              0     prevRow[ 1 ]
01ffffa8              1     prevRow[ 2 ]
```

activation record for genRow(row, 3)

```
01ffffb0  .main.enter+10    saved ra  to main
01ffffb8              0     saved fp
01ffffc0              0     saved s0
01ffffc8              0     saved s1
01ffffd0              0     saved s2
01ffffd8              0     saved s3
```

row in main

```
01ffffe0              1     row[ 0 ]
01ffffe8              0     row[ 1 ]
01fffff0              0     row[ 2 ]
01fffff8              1     row[ 3 ]
02000000                    Bottom of stack
```

12. Assembling and Disassembling

§12.1 Overview

We write a program in assembly language (or even in a high level language). This program is converted into (binary) machine code.

What is involved in this translation?

Because it is possible to refer to labels before they are declared, assemblers are usually multi-pass. My assembler is composed of the following passes:

- Lexical analysis and parsing. The input is analysed into tokens and constructs, and a tree is built, representing the structure of the program.
- Collection of declarations. A treewalk is performed, to determine the names and nesting of blocks, and the identifiers declared within each block. The mapping of block names to blocks, for the list of blocks used by a block also occurs in this pass. A consequence of this is that blocks must be declared before they are used.
- Mapping of identifiers to declarations. A treewalk is performed to map all identifier applications to identifier declarations. Essentially this pass looks up the tables generated by the previous pass.
- Address generation. A treewalk is performed to determine the offset of every statement from the base of its section, and the values of all identifiers (possibly as offsets from the base of a section). For local sections, this requires the calculation of the initial offset for the section. As a consequence, it must be defined in terms of constants and offsets of labels in previous local sections. Similarly, expressions are computed when they are needed to indicate the size of data (the expression in a space allocation statement, or an array declaration).
- Determination of the address of each section. The code and data start at addresses that depend on whether the code is PAL, kernel, or user code. The constant and global table follows immediately after the code.
- Code generation. A treewalk is performed to generate code. At this stage, all identifiers must be defined, in terms of absolute addresses.

Each pass generates errors, with the offending construct indicated, and a line number. The line number is often one line after the real error.

For example, a program generated the following error messages. A ";" was missing on line 233, which generated a syntax error, but was reported as an error on line 234. The fact that ldiq was mistyped as ldi was not picked up until address generation time, because it was only in this pass that an attempt was made to determine the opcode corresponding to its name.

```
Assembling USER file "/Home Machine/Data/ALPHACODE2.05/SIMPLE/TESTPROG ERROR/user
code.user.s" ...
Parsing ...
usercode.user.s : 40 : Syntax Error
ge1;||||bsr||print.enter;||||ldiq|$a0, |buffer||||ldiq|$a1,|BUFFERSIZE;|||
    ||||   ||            ||||    |     |      ||||^^^^|    |           |||
Generating Declarations ...
Looking Up Declarations ...
Generating Addresses ...
usercode.user.s : 47 : Invalid opcode "ldi"
ter;||||ldiq|$a0,|buffer;||||bsr||print.enter;||||ldi||$a0,|NEWLINE;||||bsr||putC
    ||||    |    |       ||||   ||            ||||^^^^^^^^^^^^^^^^^^||||   ||
Generating Code ...
Completed Generating Code ...
User Error: Error in compilation of /Home Machine/Data/ALPHACODE2.05/SIMPLE/TESTP
ROG ERROR/usercode.user.s
LoadError: User Error while loading file
```

The "|"s represent control characters such as line breaks and tabs. The "^^^^^" represent an indication of the construct in which the error occurred.

Each pass is capable of generating errors. Sometimes an error in a previous pass might cause spurious errors in subsequent passes. Equally well, an error in a previous pass might cause real errors in subsequent passes to not be detected.

Another point worth noting is that an attempt is made to load a program, even if it contains errors. The user can then attempt to run this program. You do need to look in the trace window to check whether there were any error messages.

It is not difficult to assemble and disassemble instructions.

Alpha opcodes and function codes for some common instructions

We have to know the opcode and function codes of each instruction. For example, the format, opcode and function code is indicated below for some of the common instructions.

Name	Format	Opcode	Function code
addq	Operate	0x10	0x20
subq	Operate	0x10	0x29
mulq	Operate	0x13	0x20
sra	Operate	0x12	0x3c
lda	Memory	0x8	
ldq	Memory	0x29	
ldbu	Memory	0xa	
stq	Memory	0x2d	
beq	Branch	0x39	
bne	Branch	0x3d	

Alpha Registers

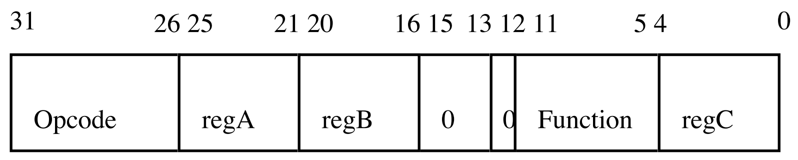
0	v0
1-8	t0-t7
9-14	s0-s5
15	fp
16-21	a0-a5
22-25	t8-t11

26	ra
27	pv
28	at
29	gp
30	sp
31	zero

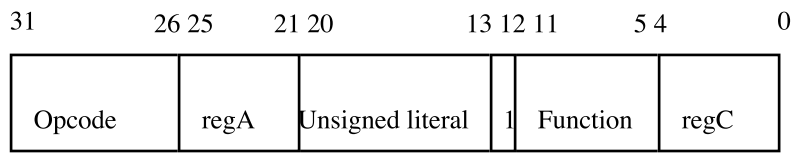
(Note: The register numbers are in decimal).

§12.2 Integer operate instructions

Integer operate instructions have the following format:



Integer operate instruction with second operand a register



Integer operate instruction with second operand a literal

Suppose we have the instruction "addq $a0, $t0, $t2;". The identifiers a0, t0, t2 are symbolic names for registers 16, 1 and 3 (decimal), so we could write the instruction as "addq $16, $1, $3;". Moreover, the literal flag must be 0, so the fields for the instruction are:

Field	opcode	regA	regB	padding	literal flag	function	regC
Hex	0x10	0x10	0x1	0x0	0x0	0x20	0x3
Binary	010000	10000	00001	000	0	0100000	00011

Grouping the bits in lots of 4 we get

0100 0010 0000 0001 0000 0100 0000 0011

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0x42010403.

Consider the instruction "subq $t5, 1;". Expanding this out to three operands, and replacing the symbolic name, we get "subq $6, 1, $6;". We have an integer operate format, with a literal for the second operand, so the fields for the instruction are:

Field	opcode	regA	literal value	literal flag	function	regC
Hex	0x10	0x6	0x1	0x1	0x29	0x6
Binary	010000	00110	00000001	1	0101001	00110

Grouping the bits in lots of 4 we get

0100 0000 1100 0000 0011 0101 0010 0110

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0x40c03526.

The computer must perform the translation in reverse order. Given the instruction in internal form, it must be able to determine the opcode and operands, so that it can execute the instruction. For example, suppose we have an instruction 0x4cf5540e.

Writing this in binary, we get

0100 1100 1111 0101 0101 0100 0000 1110.

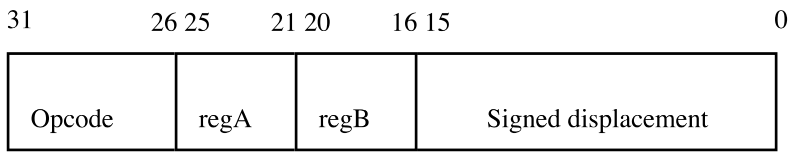
Now the 6 bit opcode is 010011, namely 0x13, which represents an integer operate instruction. Moreover bit 12 is 1, so the instruction has a literal for the second operand. Splitting it up into its fields, we get

Field	opcode	regA	literal value	literal flag	function	regC
Binary	010011	00111	10101010	1	0100000	01110
Hex	0x13	0x7	0xaa	0x1	0x20	0xe
Decimal		7	170			14

Now opcode 0x13, and function code 0x20 represents the mulq instruction. So we must have the instruction “mulq $7, 170, $14;”, or using symbolic names for registers, “mulq $t6, 170, $s5;”.

§12.3 Memory access instructions

Memory access instructions have the following format



Memory access instruction

The displacement is a signed two’s complement number.

Suppose we have the instruction “lda $sp, +10($sp);”.

We get

Field	opcode	regA	regB	displacement
Hex	0x8	0x1e	0x1e	0xa
Binary	001000	11110	11110	0000000000001010

(Remember that decimal 10 is hexadecimal 0xa and binary 1010.)

Grouping the bits in lots of 4 we get

0010 0011 1101 1110 0000 0000 0000 1010

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0x23de000a.

Suppose we have the instruction “lda $sp, -10($sp);”.

We get

Field	opcode	regA	regB	displacement
Hex	0x8	0x1e	0x1e	0xfff6
Binary	001000	11110	11110	1111111111110110

(We can represent decimal -10 as a two’s complement number by writing decimal 10 in binary as 0000000000001010, taking the one’s complement 1111111111110101, then adding 1 to get 1111111111110110. We have to take into account the number of bits used to store the value.)

Grouping the bits in lots of 4 we get

0010 0011 1101 1110 1111 1111 1111 0110

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0x23defff6.

Assembling and Disassembling

Suppose we have the instruction 0x23deffe0. In binary this is:

0010 0011 1101 1110 1111 1111 1110 0000

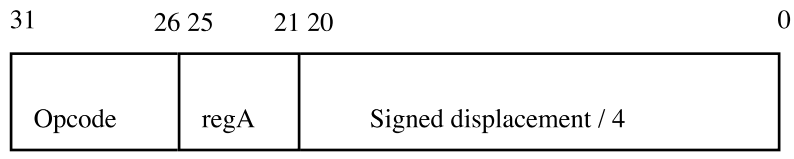
The opcode is 0x8, so we again have a lda instruction. Splitting it up into fields, we get

Field	opcode	regA	regB	displacement
Hex	0x8	0x1e	0x1e	0xffe0
Binary	001000	11110	11110	1111111111100000

In other words, “lda $sp, -0x20($sp);”. (We can determine the negative number the displacement corresponds to by taking the two’s complement, to get a positive number. Alternatively, we can subtract 0xffe0 from 0x10000.)

§12.4 Branch instructions

Branch instructions are a little more complex, because the displacement stored in the instruction is relative to the program counter, at the time at which the instruction is executed (after the program counter has been incremented to point to just after the instruction), and the displacement is counted in longwords (in other words, the low two bits of the byte displacement are discarded), because all instructions must be longword aligned.



Branch instruction

Suppose we have an instruction “bne $s1, label1;”, at address 0x80023c, and label1 correponds to address 0x80027c.

The program counter will be 0x800240 at the time the instruction is executed. So the address to branch to is 0x80027c - 0x800240 = +0x3c bytes away. Dividing this by 4 (the size of a longword) gives us a displacement of +0xf. The opcode for bne is 0x3d, and register s1 is register 10 (decimal), so we get:

Field	opcode	regA	displacement
Hex	0x3d	0xa	0xf
Binary	111101	01010	000000000000000001111

Grouping the bits in lots of 4 we get

1111 0101 0100 0000 0000 0000 0000 1111

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0xf540000f.

Suppose we have an instruction “beq $v0, label2;” at address 0x80025c, and label2 correponds to address 0x80022c.

The program counter will be 0x800260 at the time the instruction is executed. So the address to branch to is 0x80022c - 0x800260 = -0x34 bytes away (0x7fffcc, when written as a 23 bit unsigned number). Dividing this by 4 (the size of a longword) gives us a displacement of -0xd (0x1ffff3, when written as a 21 bit unsigned number). The opcode for beq is 0x39, and register v0 is register 0, so we get:

Field	opcode	regA	displacement
Hex	0x39	0x0	-0xd (0x1ffff3)

Binary	111001	00000	111111111111111110011

(There are various ways of performing the arithmetic. One way is to do everything in binary. Another way is to do it in hexadecimal. Negative numbers come out as numbers with an infinite number of ..fffff on the left. When we pack the data in the displacement field, we discard the extra bits.)

Grouping the bits in lots of 4 we get

1110 0100 0001 1111 1111 1111 1111 0011

and writing it in hexadecimal, we can see that the instruction is encoded as the number 0xe41ffff3.

Suppose we have the instruction 0xe6000003, at address 0x800200. In binary this is

1110 0110 0000 0000 0000 0000 0000 0011.

The opcode is 0x39, so it is a beq instruction.

Splitting it up into fields, we get:

Field	opcode	regA	displacement
Hex	0x39	0x10	0x3
Binary	111001	10000	000000000000000000011

So the destination adress is 4 * 0x3 + 0x800204 = 0x800210, giving the instruction "beq $a0, 0x800210;". Of course, if the address 0x800210 has a label, we can replace it by the symbolic label.

Exercise ASSEMBLE

Assemble the following program. Assume the code is at address 0x800000, and the data is at address 0x1000000.

```
entry main.enter;


block main {
     data {
          aaa:
               quad 123;
          bbb:
               asciiz "\"hi\"\n";
          } data
     code {
     public enter:
               beq       $t0, yyy;
          xxx:
               ldq       $s0, ($t0);
               subq      $s0, 1;
               bne       $s0, xxx;
          yyy:
               addq $zero,    123, $t0;
               br        xxx;
          } code
     } block main
```

Instruction	Format	Opcode	Function code
beq	Branch	0x39	
bne	Branch	0x3d	
br	Branch	0x30	
ldq	Memory	0x29	
addq	Operate	0x10	0x20
subq	Operate	0x10	0x29

Appendices

13. Commonly used Alpha instructions

Integer operate instructions

```
Opcode $regA, $regB, $regC
```
intReg[regC] = intReg[regA] op intReg[regB]

```
Opcode $regA, constantB, $regC
```
The constant is an 8 bit unsigned constant.
intReg[regC] = intReg[regA] op constantB

Arithmetic integer operate instructions

addq	add	+
subq	subtract	-
mulq	multiply	*
umulh	top half of 128 bit multiply	*
divq/divqu	divide, signed/unsigned	/
modq/modqu	modulo, signed/unsigned	%
s8addq	scaled 8 add	8*operandA+operandB
S4addq	scaled 4 add	4*operandA+operandB

Shift integer operate instructions

sll	shift left logical	<<
srl	shift right logical	>>>
sra	shift right arithmetic	>>

Compare integer operate instructions

cmpeq	compare equal	==
cmplt/cmpult	compare less than signed/unsigned	<
cmple/cmpule	compare less than or equal signed/unsigned	<=

Logical integer operate instructions

and	and	&
bic	bit clear	& ~
bis/or	bit set/or	|
eqv/xornot	equivalent/exclusive or not	^ ~
ornot	or not	| ~
xor	exclusive or	^

Conditional move instructions

```
Opcode $regA, $regB, $regC
```
```
if ( relation holds for intReg[ regA ] )
    intReg[ regC ] = intReg[ regB ]
```

```
Opcode $regA, constantB, $regC
```
```
if ( relation holds for intReg[ regA ] )
    intReg[ regC ] = constantB
```

cmoveq	conditional move equal
cmovne	conditional move not equal
cmovlt	conditional move less than
cmovle	conditional move less than or equal
cmovgt	conditional move greater than
cmovge	conditional move greater than or equal
cmovlbs	conditional move low bit set
cmovlbc	conditional move low bit clear

Memory instructions

```
Opcode $regA, displacement($regB)
Opcode $regA, ($regB)
Opcode $regA, constant
```

The displacement or constant is a 16 bit signed constant.

Load address instruction

intReg[regA] = displacement + intReg[regB]

lda	load address

Load memory instructions

intReg[regA] = Memory[displacement + intReg[regB]]

ldq	load quadword
ldl	load longword
ldwu	load word unsigned
ldbu	load byte unsigned

Store memory instructions

Memory[displacement + intReg[regB]] = intReg[regA]

stq	store quadword
stl	store longword
stw	store word
stb	store byte

Branch instructions

Conditional branch instructions

```
Opcode $regA, destination
```

```
if ( relation holds for intReg[ regA ] )
        programCounter = destination
```

beq	branch equal
bne	branch not equal
blt	branch less than
ble	branch less than or equal
bgt	branch greater than
bge	branch greater than or equal
blbs	branch low bit set
blbc	branch low bit clear

Unconditional branch instructions

```
Opcode destination;
```

```
programCounter = destination          //   br

intReg[ ra ] = programCounter         //   bsr
programCounter = destination
```

br	branch
bsr	branch to subroutime

Jump instruction

```
Opcode ($regA);
```

```
programCounter = intReg[ regA ]       //   jmp

intReg[ ra ] = programCounter         //   jsr
programCounter = intReg[ regA ]
```

jmp	jump
jsr	jump to subroutine

Return instruction

programCounter = intReg[ra]

ret	return

Callpal instruction

```
call_pal constant;
```

The constant is a 26 bit constant.

call_pal	call PALcode

Pseudoinstructions

Load immediate

```
ldiq $regA, constant
```

The constant is a 64 bit constant.
intReg[regA] = constant

ldiq	load immediate quadword

Clear

```
clr $regA
```

intReg[regA] = 0

clr	clear

Unary pseudoinstructions

```
Opcode $regB, $regC
```

intReg[regC] = op intReg[regB]

```
Opcode constantB, $regC
```

The constant is an 8 bit unsigned constant.
intReg[regC] = op constantB

mov	move
negq	negate

14. System calls and library functions in the simulator

User Call PAL instructions in the simulator

- call_pal CALL_PAL_CALLSYS. System call instruction. Enter a system call.
- call_pal CALL_PAL_BPT. Breakpoint instruction. Stop, so that the program can be resumed.

To implement a system call, pass the system call number in $a0, and the arguments in $a1, $a2, $a3, ..., and return the result in $v0.

Library functions in block Sys

- int getChar(). Reads a character from the simple terminal.
- int putChar(char c). Writes a character to the simple terminal.
- void exit(int status). Causes the process to exit. You need this at the end of your main program.
- void breakpoint(). Causes the process to stop, so that it can be resumed.

Library functions in block IO

- void newline(). Prints a newline.
- void print(char *s). Prints a string.
- void error(char *s). Prints a string then exits.
- char * readLine(char *s, int max). Reads a line of input into a buffer, and terminates the text with a null byte. Discards text that will not fit into the buffer. Return the address of the null byte just beyond the end of the text. Returns null on end of file.
- void printf(char *s, int param0, int param1, int param2, int param3, int param4). Prints the parameters according to the format string s. Indicate format directive by %. Specify alignment by - (left) or + (right). Specify 0 pad character by 0 (omit for space). specify field width in decimal. %b, (binary) %o, (octal) %d (decimal), %x, (hexadecimal) %c (character), %s (string), %% (to escape %). E.g., printf(“value = %+024x\n”, value).

Library functions in block Number

- int fromString(char *buffer, int base). Converts the text in the buffer from the specified base into internal form. If base is 0, determines the base from the start of the text.
- char *toUnsigned(unsigned int value, int base). Converts the unsigned number into a base.
- char *toSigned(int value, int base). Converts the signed number into a base.

Library functions in block String

- char *fromChar(char c). Creates a string containing the character c.
- int compare(char *s, char *t). Compares two strings and indicates their order by a value <, ==, > 0 depending on whether s < t, s == t, s > t.
- int length(char *s). Returns the length of the string s.
- void copy(char *s, char *t). Copies the string pointed to by t into the buffer pointed to by s. Does not cope with overlapping strings.
- char *padLeft(char *s, char padChar, int fieldWidth). Pads s on the left with the pad character, to create a string of length fieldWidth. Trims s on the right if more than fieldWidth characters long.
- char *padRight(char *s, char padChar, int fieldWidth). Pads s on the right with the pad character, to create a string of length fieldWidth. Trims s on the left if more than fieldWidth characters long.

Functions that return a string (char *) use static space. The space will be overwritten by a later invocation.

15. Function invocation conventions

The arguments are passed in $a0, $a1, $a2, ..., and the result is returned in $v0.

Space can be allocated on the stack by subtracting the amount of space needed from the stack pointer on entry to the function. This space should be deallocated by adding the amount of space needed to the stack pointer on exit from the function. The values of registers can then be saved on the stack on function entry and restored on function exit. The invoked function may make use of the registers, during the invocation, but this use will not be visible to the invoker.

Functions that alter the stack pointer register $sp must restore it on return. In other words, the amount subtracted from the stack pointer on function entry and the amount added to the stack pointer on function exit must agree.

Functions must save and restore any “saved” registers $s0, $s1, $s2, ..., that they make use of. Hence a function invocation will not appear to alter any of the saved registers.

The bsr instruction saves the program counter in the return address register $ra. Hence the return address register will be altered by a function invocation. If a function invokes another function, the invoker must save and restore the $ra register.

Functions can modify the “temporary” registers $t0, $t1, $t2, ..., wiithout saving and restoring them. Similarly functions can modify the argument registers, and $v0. Hence the invoker cannot assume data in these registers will remain on return from the function.

16. Handling of Exceptions and Interrupts in the Simulator

When an exception or interrupt occurs, PAL mode is entered at an address specified by the PAL exception/interrupt vector table.

The cause of the exception or interrupt can be

• RESET	Machine reset. This effectively occurs when the machine is turned on. The machine starts executing the RESET handler.
• MCHK	Machine check. This should represent a hardware failure, and should never happen.
• ARITH	An arithmetic exception (integer overflow, floating point overflow, floating point underflow, inexact floating point result, divide by zero, invalid operand).
• INTERRUPT	Any interrupt (clock, disk, keyboard, screen, or software interrupt).
• D_FAULT	A data access fault (fault on read/write, access control (protection) violation (e.g., trying to write to read-only memory), non-existent physical memory for a load or store). For naive users of the simulator this usually means the virtual address being accessed by a load or store instruction has the wrong protection or does not exist.
• ITB_MISS	An instruction translation buffer miss (page table entry not in the instruction translation buffer). For naive users of the simulator this usually means the virtual address being accessed by the program counter does not exist or does not correspond to code.
• ITB_ACV	An instruction fetch fault (access control (protection) violation (e.g., trying to execute non-executable memory), non-existent physical memory for an instruction fetch).
• DTB_MISS_NATIVE	A data translation buffer miss (page table entry not in the data translation buffer), when not in PAL mode. For naive users of the simulator this usually means the virtual page being accessed by a load or store instruction does not exist.
• DTB_MISS_PAL	A data translation buffer miss (page table entry not in the data translation buffer), when in PAL mode. This is generated if the page table entry for the kernel stack is not in the data translation buffer when attempting to modify the kernel stack on entry to or exit from an exception.
• UNALIGN	Unaligned access fault (a load or store instruction is trying to access memory at an address not divisible by the size of the date being loaded or stored).
• OPCDEC	An attempt to execute an unimplemented or illegal instruction. For naive users of the simulator this usually means an attempt to execute data as code. For example, they might have flowed into the constant section or global table, because they missed out an instruction to invoke Sys.exit.
• FEN	An attempt to execute a floating point instruction when the floating point instruction flag is not enabled.

• CALL_PAL_KERNEL	Execution of a call_pal instruction from kernel mode. This is often caused by the execution of the retsys instruction, and does not represent an error.
• CALL_PAL_USER	Execution of a call_pal instruction from user mode, including system calls. This is often caused by the execution of the callsys instruction, and does not represent an error.

17. The Alpha Assembler Lexical and Syntactic Structure

§17.1 Lexical Structure

Layout

In this assembler, programs are essentially free format, in the sense that blanks, tabs, carriage returns, and line feeds are largely irrelevant. Unlike most assemblers, statements can be split over multiple lines. It does not matter whether line breaks are represented by CR, LF, or CR/LF pairs, so it does not matter what kind of machine you are using. ";"s take the place of line breaks, in that they are used to terminate simple statements.

Comments

There are two kinds of comments. Single line comments are of the form

```
    // Text until end of line
```

Multi-line comments are of the form

```
/*
Possibly multi line text
*/
```

/* */ style comments can be nested.

Literals

The assembler allows decimal, octal, and hexadecimal integer literals, floating point literals, string and character literals. All have much the same format as in C and Java.

Zero is represented by `0`.

Octal integers are of the form `0[0-7]*`.

Decimal integers are of the form `[1-9][0-9]*`.

Hexadecimal integers are of the form `0[xX][0-9A-Fa-f]+`.

Thus unless an integer starts with a `0` or `0x`, it is interpreted as decimal.

Floating point numbers are of the form `{head}{tail} | {head}{exp} | {head}{tail}{exp}`, where `head` represents `[0-9]+`, `tail` represents `[.]{digit}+`, `sign` represents `[\+\-]?`, and `exp` represents `[eE]{sign}[0-9]+`.

Character literals are of the form `\'{chr}\'`, and string literals are of the form `\"{chr}*\"`, where `chr` represents `[^\"\r\n\\]|\\{escape}`, `escape` represents `{octalesc} | {hexesc} | {charesc}`, `octalesc` represents `[0-7] | [0-7][0-7] | [0-3][0-7][0-7]`, `hexesc` represents `[xX]{hexdigit} | [xX]{hexdigit}{hexdigit}`, `charesc` represents `[ntbrf\\\"\']` (linefeed, tab, backspace, carriage return, form feed, backslash, double quote).

Identifiers

Identifiers are of the form [A-Za-z_][A-Za-z0-9_]*, in other words a letter, followed by zero or more letters or digits. An underscore is considered to be a letter.

Keywords

The assembler has the following keywords:

`entry`, `extends`, `uses`, `abs`, `code`, `const`, `data`, `local`, `block`, `public`, `private`, `protected`, `align`, `ascii`, `asciiz`, `byte`, `ubyte`, `word`, `uword`, `long`, `ulong`, `quad`, `uquad`, `float`, `double`, `space`, `enclosing`.

Special symbols

The assembler has the following special symbols:

“;”, “:”, “,”, “$”, “(”, “)”, “[”, “]”, “{”, “}”, “+”, “-”, “*”, “/”, “%”, “<<”, “>>”, “>>>”, “^”, “|”, “&”, “~”, “=”, “.”.

§17.2 Syntactic Structure

Program

```
Program ::=
         EntryOpt
         InitStmtSeq
     ;

EntryOpt ::=
         /* Empty */
     |
         "entry" Expr ";"
     ;
```

Sections

```
SectionSeq ::=
         /* Empty */
     |
         SectionSeq Section
     ;

Section ::=
         "code"
         "{"
         LabelledInitStmtSeq
         "}"
         "code"
     |
         "const"
         "{"
         LabelledInitStmtSeq
         "}"
         "const"
     |
         "data"
         "{"
         LabelledInitStmtSeq
         "}"
         "data"
     |
         "local"
         "{"
         LabelledUninitStmtSeq
         "}"
         "local"
     |
         "abs"
         "{"
         AbsStmtSeq
         "}"
         "abs"
     |
         "block" IDENT ExtendsOpt UsesOpt
         "{"
         SectionSeq
```

```
          "}"
          "block" IDENT
     |
          Access "block" IDENT ExtendsOpt UsesOpt
          "{"
          SectionSeq
          "}"
          "block" IDENT
     |
          "import" STRINGCONST ";"
     ;

ExtendsOpt ::=
          /* Empty */
     |
          "extends" Expr
     ;

UsesOpt ::=
          /* Empty */
     |
          "uses" NameSeq
     ;
```

Statement Sequences

```
LabelledInitStmtSeq ::=
          InitStmtSeq
     |
          InitStmtSeq
          EndLabelStmt
     ;

LabelledUninitStmtSeq ::=
          UninitStmtSeq
     |
          UninitStmtSeq
          EndLabelStmt
     ;

InitStmtSeq ::=
          /* Empty */
     |
          InitStmtSeq InitStmt
     ;

UninitStmtSeq ::=
          /* Empty */
     |
          UninitStmtSeq UninitStmt
     ;
AbsStmtSeq ::=
          /* Empty */
     |
          AbsStmtSeq AbsStmt
     ;
```

End Label Statements

```
EndLabelStmt ::=
          IDENT ":"
     |
          Access IDENT ":"
     |
          IDENT ":" EndLabelStmt
     |
        Access IDENT ":" EndLabelStmt
    ;
```

Initialised Statements

```
InitStmt ::=
        IDENT OperandSeqOpt ";"
    |
        Type Expr ";"
    |
        Type SizeSeq Expr ";"
    |
        IDENT ":" InitStmt
    |
        Access IDENT ":" InitStmt
    |
        "align" ";"
    |
        "align" Type ";"
    |
        Type ";"
    |
        Type SizeSeq ";"
    |
        "space" Expr ";"
    |
        "space" Expr SizeSeq ";"
    |
        Access IDENT
        "{"
        LabelledInitStmtSeq
        "}"
        IDENT
    |
        IDENT
        "{"
        LabelledInitStmtSeq
        "}"
        IDENT
    |
        "{"
        LabelledInitStmtSeq
        "}"
    |
        ";"
    |
        Section
    ;
```

Uninitialised Statements

```
UninitStmt ::=
          IDENT ":" UninitStmt
     |
          Access IDENT ":" UninitStmt
     |
          "align" ";"
     |
          "align" Type ";"
     |
          Type ";"
     |
          Type SizeSeq ";"
     |
          "space" Expr ";"
     |
          "space" Expr SizeSeq ";"
     |
          Access IDENT
          "{"
          LabelledUninitStmtSeq
          "}"
          IDENT
     |
          IDENT
          "{"
          LabelledUninitStmtSeq
          "}"
          IDENT
     |
          "{"
          LabelledUninitStmtSeq
          "}"
     |
          ";"
     |
          Section
     ;
```

Absolute Statements

```
AbsStmt ::=
          IDENT "=" Expr ";"
     |
          Access IDENT "=" Expr ";"
     |
          ";"
     |
          Section
     ;
```

Access Modifiers

```
Access ::=
          "public"
     |
          "private"
     |
          "protected"
     ;
```

Types

```
Type ::=
         "ascii"
     |
         "asciiz"
     |
         "byte"
     |
         "ubyte"
     |
         "word"
     |
         "uword"
     |
         "long"
     |
         "ulong"
     |
         "quad"
     |
         "uquad"
     |
         "float"
     |
         "double"
     ;
```

Array Size Sequence

```
SizeSeq::=
         /* Empty */
     |
         SizeSeq "[" Expr "]"
     ;
```

Operands

```
OperandSeqOpt ::=
         /* Empty */
     |
         OperandSeq
     ;

OperandSeq ::=
         Operand
     |
         OperandSeq "," Operand
     ;

Operand ::=
         "$" Expr
     |
         Expr
     |
         "(" "$" Expr ")"
     |
         Expr "(" "$" Expr ")"
     ;
```

Expressions

```
Expr ::=
         Expr "+" Term
     |
         Expr "-" Term
     |
         Term
     ;

Term ::=
         Term "*" Factor
     |
         Term "/" Factor
     |
         Term "%" Factor
     |
         Term "<<" Factor
     |
         Term ">>" Factor
     |
         Term ">>>" Factor
     |
         Term "^" Factor
     |
         Term "|" Factor
     |
         Term "&" Factor
     |
         Factor
     ;

Factor ::=
         "+" Factor
     |
         "-" Factor
     |
         "~" Factor
     |
         "(" Expr ")"
     |
         "(" Type ")" Factor
     |
         OCTINTCONST
     |
         DECINTCONST
     |
         HEXINTCONST
     |
         CHARCONST
     |
         FLOATCONST
     |
         STRINGCONST
     |
         Name
     ;
```

Names

```
NameSeq ::=
         Name
     |
         NameSeq "," Name
     ;

Name ::=
          "enclosing"
     |
          IDENT
     |
          Name "." IDENT
     ;
```

§17.3 Programs, sections and blocks

An assembly language program starts with an optional entry point specification (default, start of the code section), followed by a sequence of statements, which are usually sections, import directives and blocks.

```
entry main.enter;

import "../IMPORT/callsys.h";

//   void main() {
//        while ( TRUE ) {
//             char c;
//             c = getChar();
//             putchar( c );
//             }
//        }
block main uses CALLSYS {
     code {
     public enter:
     loop:
          ldiq $a0,      CALLSYS_GETCHAR;
          call_pal       CALL_PAL_CALLSYS;
          mov            $v0,      $a1;
          ldiq $a0,      CALLSYS_PUTCHAR;
          call_pal       CALL_PAL_CALLSYS;
          br        loop;
     end:
          } code
     } block main
```

So in the above program, the entry point is the label enter, within the block main. The code in the file "`../IMPORT/callsys.h`" is imported. The syntax for an imported code file is the same as for the main program, except that an entry point should not be specified. Files must not be imported more than once. File path names should be given in UNIX format.

An absolute section contains declarations of symbolic names for constants. Using symbolic names provides a way of making our programs easy to read. For example, we can declare symbolic names for registers.

A code section is normally composed of instructions to execute. The block corresponding to a function will contain a code section.

A const section is composed of the data for string constants, etc., that will not be altered.

A data section is composed of the data for global variables, that might be altered.

```
//   char buffer[ BUFFERSIZE + 1 ];
//   void main() {
//        while ( TRUE ) {
//             print( "Type some input: " );
//             readline( buffer, BUFFERSIZE );
//             print( "The input was: " );
//             print( buffer );
//             newline();
//             }
//        }
block main uses proc {
     abs {
          NEWLINE        =    '\n';
          BUFFERSIZE     =    200;
          } abs
     const {
     message1:
          asciiz    "Type some input: ";
     message2:
          asciiz    "The input was: ";
          } const
     data {
     buffer:
          byte [ BUFFERSIZE + 1 ];
          } data
     code {
     public enter:
          {
          loop:
               ldiq $a0, message1;
               bsr       IO.print.enter;
               ldiq $a0, buffer;
               ldiq $a1, BUFFERSIZE;
               bsr       IO.readLine.enter;
               ldiq $a0, message2;
               bsr       IO.print.enter;
               ldiq $a0, buffer;
               bsr       IO.print.enter;
               bsr       IO.newline.enter;
               br        loop;
          end:
               }
          } code
     } block main
```

A local section is used to define the offsets for fields of records, activation records of functions, etc. It does not allocate static space. It is primarily used to generate the values of symbols representing the offsets of fields from the base of the record, and the offsets of saved registers and local variables from the base of the activation record.

```
block proc {
     local {
          protected savRet:  quad;
          protected savFP:   quad;
          protected sav0:    quad;
          protected sav1:    quad;
          protected sav2:    quad;
          protected sav3:    quad;
          protected sav4:    quad;
          protected sav5:    quad;
          protected sav6:    quad;
          } local
     } block proc
```

The Alpha Assembler Lexical and Syntactic Structure

A block is a named compound object, composed of sections, sub-blocks, etc. A block is often used to contain all the code for a function. A class declaration might be represented by a block containing sub-blocks for functions declared within the class. We can also use a block just to group related constants together.

If a block specifies it extends a value, it means that the offsets for its local section start from that value. It is often used to specify the structure of an activation record, that allocates additional space for local variables, or additional saved registers, beyond those normally saved.

```
//   void print( char *s ) {
//        while ( *s != 0 ) {
//             putChar( *s );
//             s++;
//             }
//        }
//
public block print uses proc {
     abs {
          s         =    s0;
          } abs
     code {
     public enter:
          lda       $sp, -sav1($sp);
          stq       $ra, savRet($sp);
          stq       $s0, sav0($sp);
     body:
          mov       $a0, $s;                //   Pointer to char in string
          {
          while:
               ldbu      $a0, ($s);         //   Get character
               beq       $a0, end;          //   Break if at end of string
          do:
               bsr       Sys.putChar.enter; //   Print char
               addq      $s,  1;            //   Increment pointer
               br        while;
          end:
          }
     return:
          ldq       $s0, sav0($sp);
          ldq       $ra, savRet($sp);
          lda       $sp, +sav1($sp);
          ret;
          } code
     } block print
```

Sections within a block may be interleaved. We may specify an absolute section, a data section, a code section, then another absolute section, data section and code section. The assembler reshuffles the sections, so that the memory image created contains the complete code section, complete constant section, complete global table section, then complete data section, in that order. It is even possible to specify a constant section inside a code section. This kind of thing is useful if you want the definition of a string constant to be close to its application.

Similarly, blocks may be interleaved, and the assembler reshuffles the partial blocks, to put them together. Sometimes this feature is useful when generating assembly language using a compiler.

§17.4 Statements

There are a number of different kinds of statement.

- Instruction statements. These are used to write the instructions that make up our assembly language program.

- Label declaration statements. These are used to name memory or local offsets, so that we can refer to them. A label declaration statement contains another statement as a substatement.
- Identifier definition statements. These are used to name constant values. The constant values are computed at the time of assembly, not at run time.
- Memory allocation statements. These are used to allocate and possibly initialise memory for global and local variables.
- Compound statements. These are used to group statements together, and provide a local scope for labels. They are particularly useful when building up control statements such as loops and if statements, in that each compound statement can have its own labels with standard names, such as `while`, `do`, `end` for while loops, `if`, `then`, `else`, `end`, for if statements.
- Null statements. These are used to allow a "`;`" to be placed after a label. They are not really necessary.

Most statements, apart from compound statements are terminated by a ";".

Instructions

An instruction statement is composed of an identifier representing the opcode, followed by a comma separated sequence of operands, then a "`;`". For example:

```
    bsr      getChar.enter;     //   Get a char
    cmpeq    $v0, NEWLINE,  $t0; //   Break if newline
    blbs     $t0, end;
```

Instructions statements usually only occur within a code section.

Operands can be of the form

- `$ Expr`, to represent a register operand.
- `Expr`, to represent a literal operand, or destination in a branch instruction.
- `( $ Expr )` to represent a memory access, with zero displacement from a register.
- `Expr ( $ Expr )` to represent a memory access, with displacement from a register.

For example in

```
    cmpeq    $v0, NEWLINE,  $t0;
```

`$v0` and `$t0` represent registers, and `NEWLINE` represents a literal.

In

```
    bsr      getChar.enter;
```

getChar.enter represents the destination of a branch instruction.

In

```
    stb      $v0, ($t0);
```

`($t0)` represents displacement addressing, with a zero displacement.

In

```
    lda      $sp, -sav3($sp);
    stq      $ra, savRet($sp);
```

`-sav3($sp)` and `savRet($sp)` represent displacement addressing.

The assembler is more restrictive than most conventional assemblers. It checks that operands are within range. For example, the literal for an operate instruction must be in the range `0x0` ... `0xff` (an 8 bit unsigned literal).

There are some instructions supported by the assembler and simulator that do not exist on the real machine:

- The `divl`, `divlu`, `divq`, `divqu`, `modl`, `modlu`, `modq`, and `modqu` instructions. These instructions perform integer division and modulo arithmetic. They have the operate instruction format.
- The `call_xfc` (extended function call) instruction. This instruction is used to implement special features of the simulator, such as simple input/output, window management, etc. This instruction has the same format as a `call_pal` instruction.

There are some instructions that are special, and can only be executed in PAL mode.

- The `hw_ld`, and `hw_st` instructions. These instructions can only be used in PAL mode, and are used to load from and store to physical addresses. They have the memory instruction format.
- The `hw_mfpr`, and `hw_mtpr` instructions. These instructions can only be used in PAL mode, and are used to load from and store to special registers. They have the memory instruction format.
- The `hw_rei` instruction. This instruction can only be used in PAL mode, and is used to return from PAL mode. It has no operands.

There is very little support for pseudoinstructions (things that look like real instructions, but are translated into one or more different instructions). Maybe some additional pseudoinstructions will need to be added in later, but at the moment they are:

- The `ldiq` (load immediate quadword) and `ldit` (load immediate tfloat) pseudoinstructions. These pseudoinstructions load a constant into a register. The literal is actually stored in a table, called the global table, pointed to by a register called the gp (global pointer) register. The pseudoinstruction is actually replaced by a ldq or ldt instruction, that loads the literal value from this table, using an offset from the gp register. The `ldiq` instruction is essentially the only way you can load the address of a variable into a register.
- The `mov` (move), `negq` (negate quadword), and `not` pseudoinstructions (and similar instructions for moving or negating long or floating point values). These pseudoinstructions move, negate, or complement the value of a register, or 8 bit unsigned literal, and store it in another register. They are actually implemented by the `addq`, `subq`, and `ornot` instructions, with a zero first operand. `Mov` and `negq` can be used to load 8 bit positive and negative integers into a register. The `lda` instruction can be used to load 16 bit signed integers into a register. Large values are best loaded by the `ldiq` instruction.
- The `clr` (clear) pseudoinstruction, and a similar instruction for clearing a floating point register. This pseudoinstruction clears (zeroes) a register. It is actually implemented by a bis instruction, with the first two operands zero.

There are some real instructions for which some operands may be omitted.

- The operate instructions. The destination register may be omitted, and defaults to the first source register. Some operate instructions ignore some operands (for example, sign extension instructions). These operands should be omitted in the assembly language.
- The floating operate instructions. The destination register may be omitted, and defaults to the first source register. Some floating operate instructions ignore some operands (for example, conversion instructions). These operands should be omitted in the assembly language.

- The `jsr` (jump subroutine) and `jsr_coroutine` (jump coroutine) instructions. The saved pc register may be omitted, and defaults to the `$ra` register. The register indicating the address to jump to may be omitted for the `jsr_coroutine` instruction, and defaults to the `$ra` register.
- The `ret` (return) instruction. Both registers may be omitted. The saved pc register defaults to `$zero`, and the register indicating the address to jump to default to `$ra` register.
- The `br` and `bsr` instructions. The destination register may be omitted, and defaults to `$zero` and `$ra`, respectively.

The assembler does not support rounding or trapping flags, apart from integer overflow flags in integer instructions.

Label declaration statements

A label identifier can be declared by writing the identifier, then a colon, followed by a substatement.

For example, in
```
if:
     cmplt     $cnt,     $size,    $t0;      //   If within buffer
     blbc      $t0,      end;
then:
     addq      $ptr,     $cnt,     $t0;      //   Store the character
     stb       $v0,      ($t0);
end:
```

we declare three labels, `if`, `then` and `end`. In fact the labels `if` and `then` are only there for cosmetic reasons, to give the appearance of an if statement. They are never used.

Note that labels represent addresses (for code, constant , and data sections) or offsets (for local sections).

A label is aligned to the address of the start of the substatement. Thus, if you write
```
label:
     quad 4;
```

there is no need to precede the label by an alignment statement. If any padding needs to be allocated to align the substatement, it will occur before the label. This is a change from the way the assembler worked in the year 2001.

In the unlikely event that you actually want the padding to occur after the label, and before the substatement, append a “;” after the “:”.
```
label:
     ;
     quad 4;
```

Identifier definition statements

It is also possible to declare an identifier by an identifier definition statement.

For example
```
     ptr       =    s0;
     size      =    s1;
     cnt       =    s2;
     NEWLINE   =    '\n';
```

The expression on the right is evaluated by the assembler, and assigned to the identifier. So for example, `ptr` above has the value 9, because `s0` represents register 9 (specified in the block called register). It is important to realise that this is not a run-time action. It is not copying the contents of register `$s0`.

Identifier definition statements are used to give symbolic names to expressions, and make your code more readable.

Scope of identifiers

Identifiers can be declared as having public, protected, or private access. The default access is private.

An identifier declared within a section of a block can be referred to by its simple name anywhere within the block, including sub-blocks and compound statements.

An identifier declared within a block or named compound statement as public can be accessed outside the block or compound statement, by prefacing it by the name of the block or compound statement. Private and protected identifiers cannot be referred to in this manner.

An identifier declared within another block as public or protected, can be referred to in a block that uses it, by its simple name. Private identifiers cannot be referred to in this manner.

Unlike Java, at the moment it is not possible to refer to an identifier, declared as protected in another block, that is used by this block, by a qualified name. This may change, to become compatible with Java.

Local declarations take precedence over declarations in enclosing blocks, enclosing compound statements, and used blocks. However, if there is an ambiguity with regard to the meaning of an identifier, that is not declared locally, then an error is generated.

Memory allocation statements

We can initialise memory, by specifying a data type, followed by the initial value, then a "`;`".

```
const {
message1:
     asciiz    "Type some input: ";
message2:
     asciiz    "The input was: ";
     } const
```

Data types can be keywords such as `byte`, `ubyte`, `quad`, `ascii`, `asciiz`, etc, to allocate space for a signed byte, unsigned byte, signed quadword, unterminated ASCII string, null terminated ASCII string, etc.

Apart from the data types corresponding to strings, memory allocation instructions allocate the appropriate amount of memory in the relevant section (1 byte for `byte` and `ubyte`, 2 bytes for `word` and `uword`, 4 bytes for `long` and `ulong`, 8 bytes for `quad` and `uquad`, 4 bytes for `float`, 8 bytes for `double`). The difference between the signed and unsigned variants is to do with checking the value is in range. For example `byte` requires a value that is between `-0x80` and `+0x7f`, while `ubyte` requires a value that is between `0` and `+0xff`. In fact there is no checking for `quad` and `uquad`.

For `ascii` the number of bytes allocated is equal to the length of the string, and the contents is the data within the string. The `asciiz` directive is similar, except an extra zero byte is allocated and added on the end.

Initialised memory statements usually only occur within a constant or data section.

If we miss out the value, we get uninitialised (zero) data.

We can allocate blocks of memory, by declaring an array:

```
data {
buffer:
     byte [ BUFFERSIZE + 1 ];
     } data
```

Uninitialised memory statements usually only occur within a data section, or local section.

It is also possible to allocate blocks of memory with a specified initial value for the elements.

Space of an arbitrary size can be allocated by the space statement. This statement can be used to allocate space for record variables.

Alignment statements can be used to round the current address or offset up to a multiple of the size of a specified type. This may be needed because data has to be aligned appropriately, for it to be accessed. Generally, it is a good idea to align data labels to quadwords, no matter what the size of the data. If labels are not at least aligned to longwords, then the memory display in the simulator will be confused.

Compound statements

It is possible to create compound statements, by enclosing them in `{ ... }`. The purpose of compound statements is to create a local scope for labels. In particular, compound statements are used when implementing control statements such as loops and if statements. We can use standard identifiers such as `while`, `do`, `end`, or `if`, `then`, `else`, `end` to label the code. We can also make our code more readable by indenting the body of the compound statement.

Compound statements can also be named, by writing `IDENT { ... } IDENT`. This is useful if we wish to refer to public labels within the compound statement. The identifiers of the opening and closing braces must match. This fact is very useful in large program, for which it can be rather difficult checking that braces are matched.

Null statements

There is also a null statement, composed of nothing but a “`;`”, for people who like to put a “`;`” after a label at the end of a section.

§17.5 Expressions

It is possible to use expressions within statements. These expressions involve literals, simple and qualified names, operators, and parentheses. Two things must be borne in mind: names evaluate to addresses, not the contents of the address, and all expressions are evaluated at assembly time, and not run time.

Literals include integer, floating point, character and string values. String literals can only be used as the initial value in `ascii` and `asciiz` memory allocation statements.

Simple identifiers can correspond to labels, in which case they represent an address or offset. They can also correspond to identifiers declared in identifier definition statements, and the names of blocks or compound statements.

We can refer to public identifiers within a block or compound statement from anywhere in which the block or compound statement can be accessed, by writing the block or compound statement name, a “`.`”, then the identifier.

We can build up expressions using operators corresponding to binary “`+`”, “`-`”, “`*`”, “`/`”, “`%`”, “`<<`”, “`>>`”, “`>>>`”, “`^`”, “`|`”, “`&`”, unary “`+`”, “`-`”, “`~`”. These are primarily used with arithmetic expressions, although binary “`+`” and “`-`” can involve addresses.

It is also possible to use casts, to trim a value down to a more restrictive type. For example, we can write `( byte ) ( ~ x )`, to take the complement of `x`, and restrict it to 8 bits.

Binary “`+`” and “`-`” have the lowest precedence, then all other binary operators, then unary operators. Parentheses can be used to override precedences.