

Computer Science 210  
**Computer Systems 1**  
2007 Semester 1  
**Lecture Notes Part 2**  
**The Programmer's View of  
Computer Hardware**

*James Goodman*



## Who Am I?

- Prof. James Goodman
- Computer Science Department
- Science Centre 303-591, 38 Princes St., City
- Office Hours: No scheduled time: drop by my office or make appointment
- E-mail: [goodman@cs.auckland.ac.nz](mailto:goodman@cs.auckland.ac.nz)

2-May-07

CS210

2

## Recommended Readings

- These notes (only after the lecture):  
<http://www.cs.auckland.ac.nz/compsci210s1t/lectures>
- Dr. Bruce Hutton's lecture notes:  
<http://www.cs.auckland.ac.nz/compsci210s1t/resources>

2-May-07

CS210

3

## Why Study Computer Organization?

- Understanding how hardware and software communicate will make you a better programmer
- Some things change; some things stay the same
  - Moore's Law vs. fundamental laws
- Appreciate the power of abstraction
  - Don't write in assembly language if you don't have to!
- "Real Programmers do it in Assembly"
  - No longer an important skill

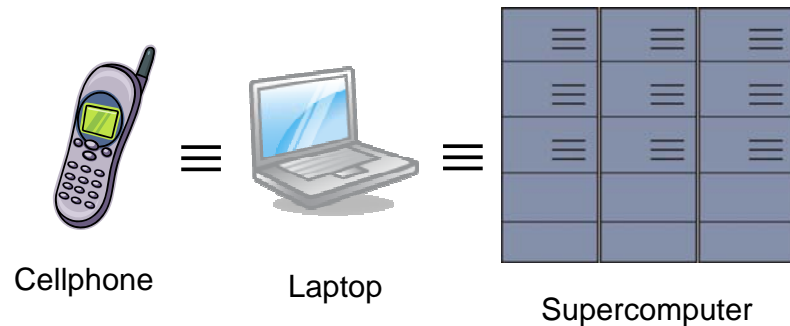
2-May-07

CS210

4

## All Computers are the Same!

- All computers, given sufficient time and memory, can compute exactly the same things.



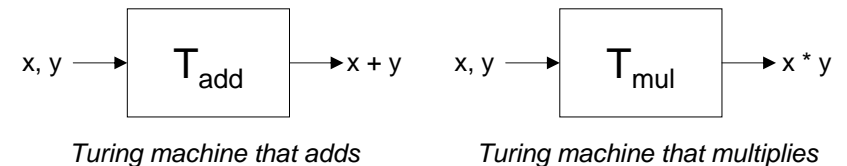
2-May-07

CS210

5

## Turing Machine

- Mathematical model of a device that can perform any computation – Alan Turing (1937)
  - ability to read/write symbols on an infinite “tape”
  - state transitions, based on current state and symbol
- Everything that can be computed can be performed by some Turing machine. (*Turing's thesis*)



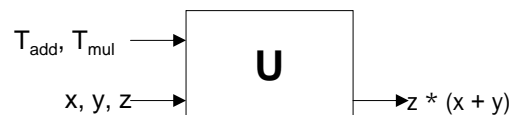
2-May-07

CS210

6

## Universal Turing Machine

- Turing described a Turing machine that could implement all other Turing machines.
  - inputs: data, plus a description of computation (Turing machine)



*Universal Turing Machine*

**U** is programmable – so is a computer!

- instructions are part of the input data
- a computer can emulate a Universal Turing Machine, and vice versa

*Therefore, a computer is a universal computing device!*

2-May-07

CS210

7

## From Theory to Practice

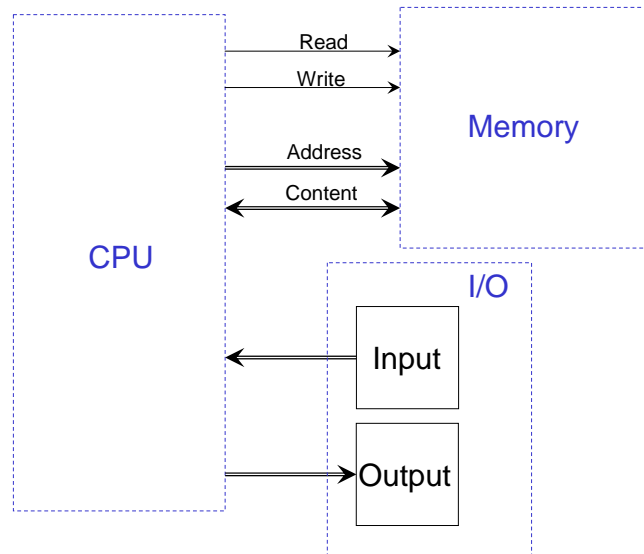
- In theory, computer can **compute** anything that's possible to compute if you
  - Have enough memory
  - Can wait long enough
- In practice, **solving problems** involves computing under constraints.
  - Time: photoshop, weather forecast, ...
  - Cost: hotel “key”, PDA, ...
  - Power: cell phone, laptop, ...

2-May-07

CS210

8

## The Von Neuman Computer

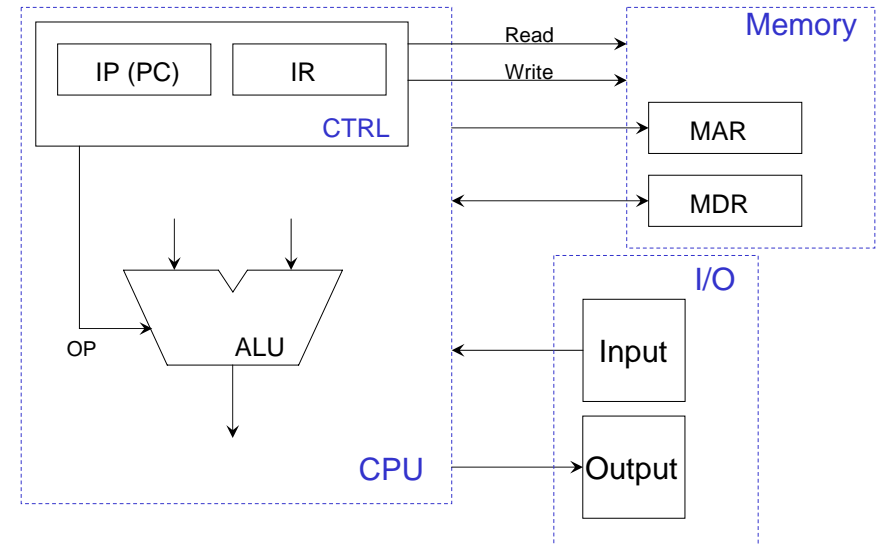


2-May-07

CS210

9

## The Von Neumann Computer



2-May-07

CS210

10

## The von Neumann Model

- Computer consists of CPU, Memory, I/O
- Memory may contain instructions or data (or meta-data)
- Does only one thing: the Instruction/Execution cycle

2-May-07

CS210

11

## The Instruction/Execution Cycle

```

Do forever {
    Fetch instruction into IR from memory address in IP
    Update IP for next instruction
    Decode instruction
    Evaluate addresses
    Fetch operands from memory
    Store result
}
    
```

2-May-07

CS210

12

## The Instruction/Execution Cycle: Variant for Control Instructions

```
Do forever {  
    Fetch instruction into IR from memory address in IP  
    Update IP for next instruction  
    Decode instruction  
    Evaluate test criterion  
    If success, store new address to PC  
}
```

2-May-07

CS210

13

## A Few Sample Instructions

Instruction	Meaning
<b>add</b> A, B, C	$C = A + B$
<b>sub</b> A, B, C	$C = A - B$
<b>mul</b> A, B, C	$C = A * B$
<b>bne</b> A, B, Label	if ( $A \neq B$ ) goto Label
<b>halt</b>	?

- A *Label* designates a memory location.
- A Label can identify either an instruction or a variable

2-May-07

CS210

14

## A Simple Program

Instructions:	Initial values:
L1: add VA, VB, VA	VA: 0
L2: sub VC, VD, VC	VB: 1
L3: mul VC, VE, VE	VC: 6
L4: bne VA, VC, L1	VD: 2
L5: halt	VE: 5
	IP: L1

2-May-07

CS210

15

## Computer Science 210 Computer Systems 1 2007 Semester 1 Lecture Notes

## The Alpha Architecture

*James Goodman*



## Recommended Readings

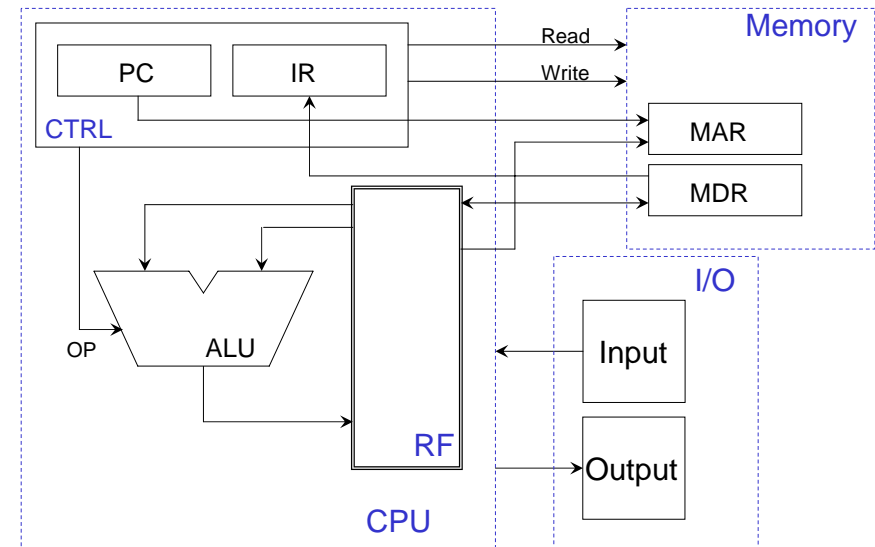
- These notes (only after the lecture):  
<http://www.cs.auckland.ac.nz/compsci210s1t/lectures>
- Dr. Bruce Hutton's lecture notes:  
<http://www.cs.auckland.ac.nz/compsci210s1t/resources>
- These lectures mostly based on chapter 2 of Dr. Hutton's notes.
- You are responsible for the first 13 chapters of Dr. Hutton's notes.
  - However, if I don't talk about it in class, it probably won't be on the exam!

2-May-07

CS210

17

## The Alpha Computer



2-May-07

CS210

18

## Four Categories of Instructions

- Arithmetic/Logical
  - Arithmetic
  - Logical
  - Shift
  - Compare
- Control
  - Branch on condition
  - Jump
    - Jump and link
- Memory: Load & Store
- Special

2-May-07

CS210

19

## Computer Science 210 Computer Systems 1 2007 Semester 1 Lecture Notes Arithmetic & Logical Instructions

*James Goodman*



## Registers

- 32 registers
- \$0 - \$31; also names
- \$31 is special
  - when read, gives zero
  - writing has no effect

2-May-07

CS210

21

## Arithmetic Instructions

- add, sub, mul (no divide)
- two sources, one destination (can be common)
- Form: **add A,B,C**
  - B can be an immediate, i.e., value contained in the instruction.
- Two operand types
  - Long word (32 bits): addl, subl, mull
  - Quad word (64 bits): addq, subq, mulq
- Overflow
  - Addition & subtraction: only one bit
  - Multiplication: up to 31 bits (additional multiplication ops)

2-May-07

CS210

22

## Logical Instructions

- Two sources, one destination
- Form: **and A,B,C**
  - B cannot be an immediate, i.e., contained in the instruction.
- One operand type: 64 bits
- Overflow: none

2-May-07

CS210

23

## Boolean Functions of 2 Variables

A	B								^		&									
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1			
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1			
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1			
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1			

Zero

NOR

ANDNOT/BIC

XOR  
XNOR

AND  
NAND

XORNOT(EQV)

OR/BIS

ORNOT

One

2-May-07

CS210

24

## Alpha Logical Operations

A	B																
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1

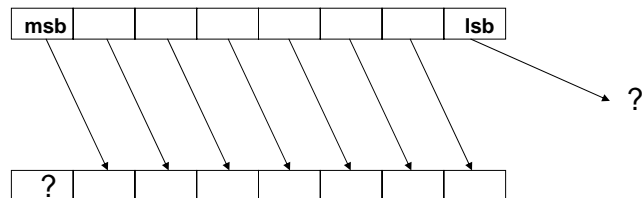
ANDNOT/BIC      XOR      AND      OR/BIS  
 XORNOT(EQV)      ORNOT

## Shift Operations

- Form: `sll A, Count, B`
- A count of  $i$  is equivalent to  $i$  shifts by 1 place.
- There are three types of Shift Operations
  - logical
  - arithmetic
  - rotate

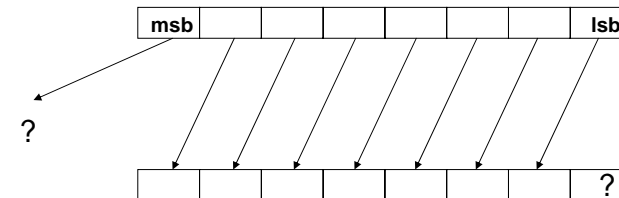
## Shift Operations

- Basic Right Shift Operation:



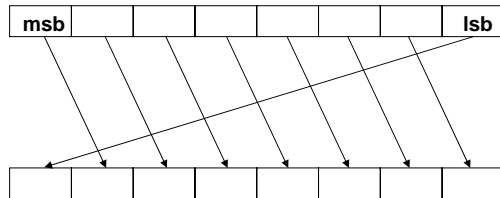
## Shift Operations

- Basic Left Shift Operation:



## Shift Operations

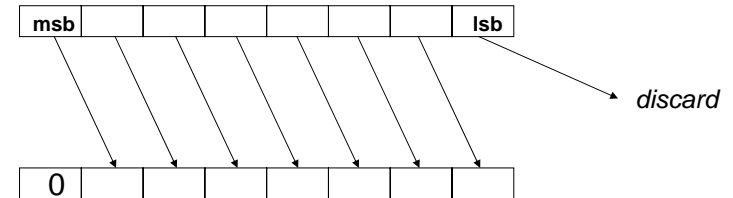
- Right Rotate Operation:



- No information lost
- For N-bit word, rotate right  $N$  positions has no effect
- Rotate right  $i$  positions is same as rotate left  $N - i$  positions
- Not implemented in Alpha (why not?)

## Logical Shift Operations

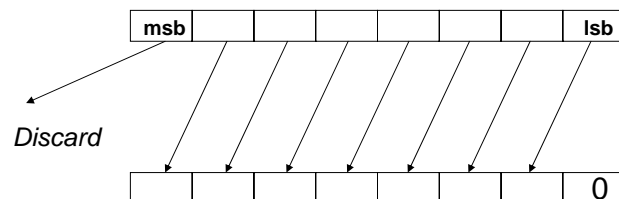
- Right Logical Shift Operation:



- Alpha instruction: **srl**
- Java equivalent: **>>>**

## Logical Shift Operations

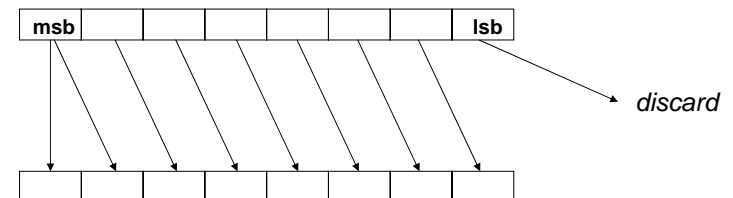
- Left Logical Shift Operation:



- Alpha instruction: **sll**
- Java equivalent: **<<**

## Arithmetic Shift Operations

- Right Arithmetic Shift Operation
  - Unsigned integer division by power of 2



- Round down (toward negative infinity)
- Alpha instruction: **sra**
- Java equivalent: **>>**
  - *same as integer division by power of 2???*



## Homework: What is -5/2 in Java?

C: "... in GNU C the '/' operator always rounds towards zero. But in other C implementations, '/' may round differently with negative arguments."

-- [http://www.gnu.org/software/libc/manual/html\\_node/Integer-Division.html](http://www.gnu.org/software/libc/manual/html_node/Integer-Division.html)

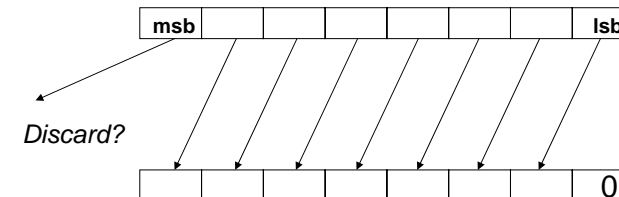
Java: "Integer division rounds toward o."

-- [http://java.sun.com/docs/books/jls/third\\_edition/html/expressions.html#15.17.2](http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.17.2)

Conclusion: In Java, '>>' is not the same operation as '/2<sup>i</sup>'

## Arithmetic Shift Operations

- Left Arithmetic Shift Operation
  - Unsigned integer multiplication by power of 2



- Overflow if MSB changes

*Same as logical left shift!*

- Alpha instruction: **sll** (no **slla**)
- Java equivalent: '**\*** 2<sup>i</sup>'

## Computer Science 210 Computer Systems 1 2007 Semester 1 Lecture Notes

## Control Instructions

James Goodman



## Control Instructions

Basic instruction for choosing alternate instruction path:

- Branch on condition (Kiw1-1): **bne VA, VC, L1**
- Alpha: **bne a, L1**
  - Register tested against zero
- Possible tests
  - beq** :  $a = 0 ?$
  - bne** :  $a \neq 0 ?$
  - bge** :  $a \geq 0 ?$
  - bgt** :  $a > 0 ?$
  - ble** :  $a \leq 0 ?$
  - blt** :  $a < 0 ?$
  - jmp** : Unconditional

# Implementing Control Structures

- While loop
- If-then-else
- For loop

## While Loop

Java:

```
while (a>0) {  
    ...  
    a--;  
}
```

Alpha Assembly:

```
WHILE: ble a, AFTERLOOP  
      sub a, 1, a  
      jmp WHILE  
      ...
```

Test for FALSE

## If-Then

Java:

```
if (a>0) {  
    ...  
}
```

Alpha Assembly:

```
IF_THEN: ble a, Continue  
          ...  
Continue: ...
```

Test for FALSE

## If-Then-Else

Java:

```
if (a>0) {  
    ...  
} else {  
    ...  
}
```

Alpha Assembly:

```
IF_THEN_ELSE: ble a, Else  
               ...  
               jmp Continue  
Else:         ...  
Continue:     ...
```

Test for FALSE

## For Loop

Java:

```
for (int i=0;i<10;i++) {  
    ...  
}
```

Alpha Assembly:

```
For:  
    add $31, 10, Limit  
    add $31, 0, i  
Loop:  
    sub Limit, i, Test  
    ble test, Continue  
    ...  
    add i, 1, i  
    jmp Loop  
Continue:  
    ...
```

2-May-07

CS210

41

## Problems with For Loop Code

Alpha Assembly:

```
For:  
    add $31, 10, Limit  
    add $31, 0, i  
Loop:  
    sub Limit, i, Test  
    ble test, Continue  
    ...  
    add i, 1, i  
    jmp Loop  
Continue:  
    ...
```

Could get overflow!

2-May-07

CS210

42

## An Alternate Control Structure

- “Arithmetic” instruction Compare:
  - `cmpeq a,b,result` if (a=b) result=1 else result = 0
  - `cmplt a,b,result` if (a < b) result=1 else result = 0
  - `cmple a,b,result` if (a>b) result=1 else result = 0
- Additional Conditional Branch instruction
  - `blbs result, L1` if (low bit of result=1) jump to L1
  - `blbc result, L1` if (low bit of result=0) jump to L1

2-May-07

CS210

43

## Computer Science 210 Computer Systems 1 2007 Semester 1 Lecture Notes

## Load/Store Instructions

*James Goodman*



## Views of Memory

- An array of bytes

000	
001	
002	
003	
004	
005	
006	
007	
008	
009	
00a	
00b	
00c	
00d	
00e	
00f	
010	
011	
...	

## Views of Memory

- An array of words

000		
002		
004		
006		
008		
00a		
00c		
00e		
010		
012		
014		
016		
018		
01a		
01c		
01e		
020		
022		
...		

## Views of Memory

- An array of longwords

000				
004				
008				
00c				
010				
014				
018				
01c				
020				
024				
028				
02c				
030				
034				
038				
03c				
040				
044				
...				

## Views of Memory

- An array of quadwords

000								
008								
010								
018								
020								
028								
030								
038								
040								
048								
050								
058								
060								
068								
070								
078								
080								
088								
...								

## Which is Correct View of Memory?

000		000		000	
001		002		004	
002		004		008	
003		006		00c	
004		008		010	
005		00a		014	
006		00c		018	
007		00e		01c	
008		010		020	
009		012		024	
00a		014		028	
00b		016		02c	
00c		018		030	
00d		01a		034	
00e		01c		038	
00f		01e		03c	
010		020		040	
011		022		044	
...		...		...	

Answer: All views are correct (and quadword too!)

## Byte Order

- Little Endian

		1	0			11	10	01	00
000		000		000		000		000	
001		002		004		004		008	
002		004		008		008		00c	
003		006		010		010		014	
004		008		014		014		018	
005		00a		018		018		01c	
006		00c		01a		01a		020	
007		00e		01c		01c		024	
008		010		01e		01e		028	
009		012		020		020		02c	
00a		014		022		022		030	
00b		016		...		...		034	
00c		018						038	
00d		01a						03c	
00e		01c						040	
00f		01e						044	
010		020						...	
011		022							
...		...							

## Byte Order

- Big Endian

		0	1			00	01	10	11
000		000		000		000		004	
001		002		004		008		00c	
002		004		008		010		014	
003		006		00c		018		01c	
004		008		010		020		024	
005		00a		014		028		030	
006		00c		016		02c		034	
007		00e		018		030		038	
008		010		01a		034		03c	
009		012		01c		038		040	
00a		014		01e		040		044	
00b		016		020		...		...	
00c		018		022					
00d		01a		...					
00e		01c							
00f		01e							
010		020							
011		022							
...		...							

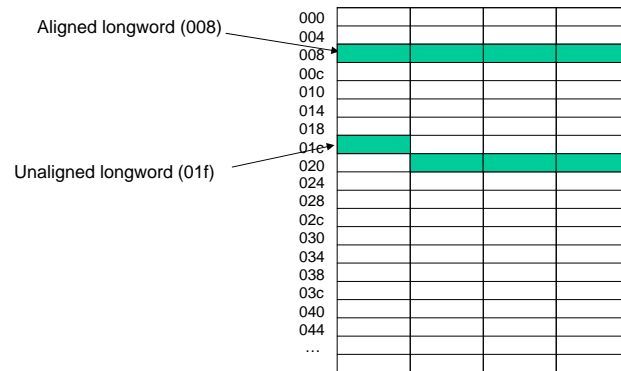
## Alignment

- An array of words (little-endian)

		000		
		002		
		004		
		006		
Aligned word (008)	→	008		
		00a		
		00c		
		00e		
		010		
		012		
		014		
		016		
		018		
		01a		
		01c		
		01e		
Unaligned word (01f)	→	01f		
		020		
		022		
		...		

## Views of Memory

- An array of longwords (little-endian)



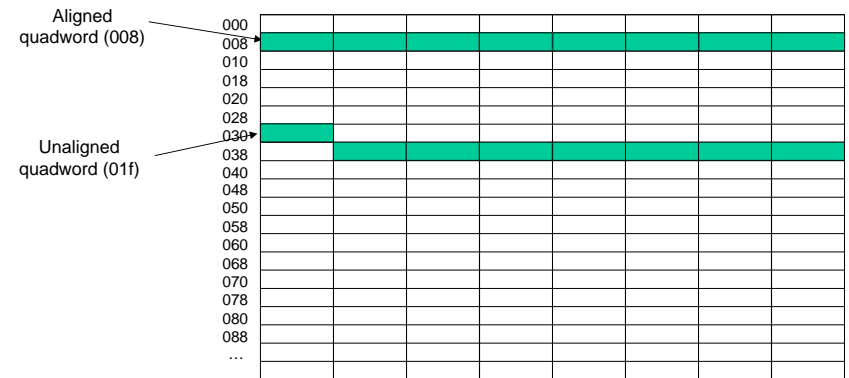
2-May-07

CS210

53

## Views of Memory

- An array of quadwords (little-endian)



2-May-07

CS210

54

## Load Instructions

- `ldq reg, ...` Load quadword
- `ldl reg, ...` Load *sign-extended* longword
- `ldwu reg, ...` Load *zero-extended* word
- `ldbu reg, ...` Load *zero-extended* byte
- `stq reg, ...` Store quadword
- `stl reg, ...` Store longword
- `stw reg, ...` Store word
- `stb reg, ...` Store byte

2-May-07

CS210

55

## Motivation: Instruction Formats

Figure 1-1: Instruction Format Overview

	31	26	25	21	20	16	15	5	4	0	
Opcode	Number										PALcode Format
Opcode	RA	Disp									Branch Format
Opcode	RA	RB	Disp								Memory Format
Opcode	RA	RB	Function	RC							Operate Format

– From The Alpha Architecture Handbook, Compaq Computer Corporation, 1998.

2-May-07

CS210

56

## Loads & Stores

**ldq Reg, Address ! Direct**

What's the problem?

- Address is stored as a constant inside the instruction
  - How to *dynamically* change the address?
- Instructions should be small, fixed size
  - Addresses are large
  - How to store a 51-bit address in a 32-bit instruction?
- Also a problem for branch instructions:

**bne Reg, Address**

## Where Do the Bits Go?

Opcode	Src1	Src2	Dest
--------	------	------	------

- Register specification requires 5 bits
- Memory specification requires up to 51 bits (at least 43)
- Opcode specification requires ?
  - 515 unique opcodes
  - 10 bits required?

## Arithmetic/Logical Instruction

Operate	Src1	Src2	Function	Dest
6 bits	5 bits	5 bits	11 bits	5 bits

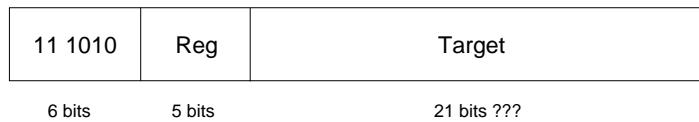
- Opcode indicates a class of instructions
  - Opcode  $10_{16}$  indicates an arithmetic function
  - Opcode  $11_{16}$  indicates a logical function
  - Opcode  $12_{16}$  indicates a shift function
- “Function” is extended Opcode, specifying which arithmetic/logical function

## Example: addq

01 0000	Src1	Src2	000 0010 0000	Dest
6 bits	5 bits	5 bits	11 bits	5 bits

- Opcode for arithmetic instructions is  $10_{16}$
- Function code for addq instruction is  $20_{16}$
- Src1, Src2 and Dest each specify 1 of 32 registers

## Example: bne



- Opcode for bne instruction is  $3d_{16}$
- Test register specifies 1 of 32 registers for testing
- Only 21 bits left for target instruction address!!!

## Idea: Add another word



- Load/store instructions could be 64 bits
- $21 + 32 = 53$  bits

But

- Instructions are not all the same size
- Load address is a constant—can't be changed within program

## Idea 2: Address is in Register

- Load instruction specifies a register to supply address: Register Indirect
- Easy to change address without changing instruction

But

- How does the address get into the register?
- How is the address modified?

## Idea 3: Construct Effective Address

- lda instruction loads constant into register  
`lda reg, constant`
- Can adjust address dynamically (using addition)
- 21-bit constant is not big enough
- Assume <constant> is 32 bits
- Break <constant> into two 16-bit pieces:  
16 most significant bits: <hi address>  
16 least-significant bits: <low address>



## Idea 3: Construct Effective Address

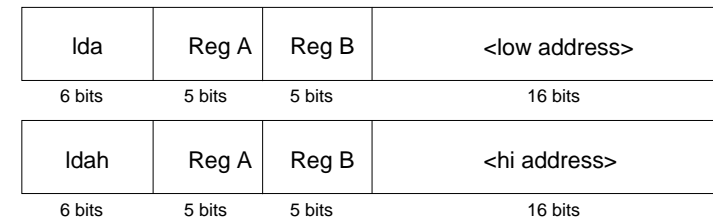
- Use instruction sequence:  
`lda $8, <hi address>`  
`sll $8, $8, 16`  
`lda $9, <low address>`  
`bis $8, $9, $10`      ! Logical OR  
`ldq $11, ($10)`      ! Register indirect
- Five instructions!
- Variant: use add instead of OR
  - sign-extend <low address>
  - `add $8, $9, $10`      ! Instead of Logical OR
  - Doesn't quite work if <low address> is negative!

2-May-07

CS210

65

## Optimization: Ida, Idah



- `lda A, <low address>(B)`
  - sign-extend constant <lowaddress> and add to contents of register B; assign result to register A
- `ldah A, <hi address>(B)`
  - multiply constant <hi address> by 65,536 and add to contents of register B; assign result to register A
- *Usually* only requires 2 instructions
  - Still only generates 32-bit addresses
  - Some 32-bit integers cannot be generated this way

2-May-07

CS210

66

## Locality of Reference

Observation: memory references are not random

- Accesses tend to be clustered
  - in time (temporal locality)
  - in space (spatial locality)
- Accesses are to objects
  - structures
  - arrays
- Can dynamic compute address arithmetically
- Can statically predict offset within known structure

2-May-07

CS210

67

## Idea 4: Use Combination

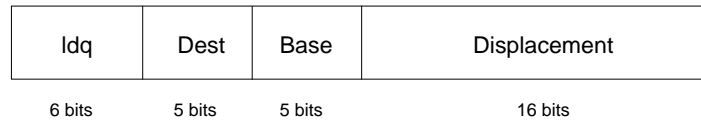
- Base + Displacement
- Base: a long-term but approximate address
  - Gives location of larger structure
  - Can be dynamically varied
- Displacement
  - Static offset embedded in instruction
  - Cannot be dynamically varied

2-May-07

CS210

68

## Load Reg, Disp(Base)



Effective address: (Base) + Displacement

- Base specifies a 64-bit address
- Displacement is a 16-bit signed constant, sign-extended to 64 bits
- Displacement defines position *relative to* Base

## Special Case of Base + Displacement

- Register Direct
  - Zero displacement
  - Register specifies address

## Load Instructions

- `ldq reg, disp(base)` ! Load quadword
- `ldl reg, disp(base)` ! Load sign-extended longword
- `ldwu reg, disp(base)` ! Load zero-extended word
- `ldbu reg, disp(base)` ! Load zero-extended byte
  
- `stq reg, disp(base)` ! Store quadword
- `stl reg, disp(base)` ! Store longword
- `stw reg, disp(base)` ! Store word
- `stb reg, disp(base)` ! Store byte
  
- `lda reg, disp(base)` ! Assign computed addr to reg
- `ldah reg, disp(base)` ! Multiply displacement by  
! 65,536 and add to base,  
! assign to address

## Summary: Possible Addressing Modes for Memory Operations

- Direct
  - address contained in instruction
- Indirect
  - Instruction contains address where address is held
- Register indirect
  - Instruction specifies register where address is held
- Register + Register
  - Instruction specifies two registers
  - Contents of registers are added to determine address
- Base + displacement
  - Instruction specifies register and contains displacement
  - Displacement is added to content of register to determine address

## Other Possible Addressing Modes

- Immediate operand
  - Instruction contains the value, used as an operand
  - Limited by word size to small constant (8 bits)
  - Example: `addq $5, 1, $5`
  - Example: `lda reg, disp($31)`
  - Example: `bne reg, target`

## Branch Instruction

- How to specify full add ( $\leq 53$  bits) in a 32-bit instruction?
- Observation: most branches are short
- Branch can use *relative address*: difference from current value of PC.

## Example: bne

11 1010	Reg	Target
6 bits	5 bits	21 bits

- Instruction must be aligned: two LSBs must be zero
- Test register specifies 1 of 32 registers for testing
- 21 bits can specify a branch relative to current instruction of  $PC \pm 2^{20}$  instructions
- New PC = PC + sign-extend( $4 * \text{Target}$ )

## Long-distance Branches

- Jump instruction
  - Full address specified indirectly through register
  - Unconditional transfer of control

## Examples of Operand Specifications

- Register (operate, control, memory)
- Unsigned 8-bit constant (operate instructions)
- Unsigned 6-bit count (shift instructions)
- Base + displacement (memory)
- 21-bit branch offset (control)
- 26-bit constant (PALcode format)

## I/O Instructions

- Privileged Architecture Library (PAL)
  - A set of functions of arbitrary complexity invoked by a special `call_pal` instruction
  - Performs privileged operations such as accessing disk, reading and printing, etc.
- Form: `call_pal` constant
  - `call_pal CALL_PAL_CALLSYS`
  - `call_pal CALL_PAL_BPT`

## Simple I/O

- `getchar` (result in `$vo`)

```
ldiq $a0, 0x1 // CALLSYS_GETCHAR
call_pal 0x83 // CALL_PAL_CALLSYS
```
- `putchar` (character in `$a1`)

```
ldiq $a0, 0x2 // CALLSYS_PUTCHAR
call_pal 0x83 // CALL_PAL_CALLSYS
```

## Computer Science 210 Computer Systems 1 2007 Semester 1 Lecture Notes Part 2

### Subroutines

*James Goodman*



## Recommended Readings

- Chapter 5: use of registers
- Chapter 11: function invocation
- Randy Bryant, *Alpha Assembly Language Guide* (available under Resources at the website) Section 3

2-May-07

CS210

81

## Registers Named

\$0	\$v0
\$1-\$8,	\$t0-\$t9
\$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 (special)

2-May-07

CS210

82

## Register Names

\$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.

2-May-07

CS210

83

## Memory Allocation for a Variable

- Global variables, constants: allocate memory permanently
  - Use registers? Maybe, if used frequently
- Local variables
  - Allocate space permanently?
    - Not needed: variables have a lifetime
    - Not sufficient: same variable might have multiple instances
  - Use registers? Likely, since they are short-lived and dynamic
- Temporary variables (used in computations)
  - Similar to local variables
  - Allocate space dynamically, probably in registers
- Arguments
  - Also have a lifetime
  - Pass in registers? Yes, if not too many
  - Also result(s), but in reverse direction

2-May-07

CS210

84

## Two Distinct Storage Issues

- Registers vs. memory
- Dynamic variables

## Dynamic Variables

Variables have a lifetime

- A variable is defined within a scope
- Variables do not need space allocated if they aren't assigned a value
- Different variables can be assigned to the same memory location at different times
- The same variables in different instances requires two different memory locations if they overlap (recursion)

## Extreme Case: Write-once variables

- A variable requires storage when it is written
- A variable does not require storage if it will not be read again before it is written
- If we know a variable will not be read, we can deallocate storage on the last read, allocate it on write.
  - We must be certain that the variable will not be read again
  - This is often possible in controlled situations, e.g., loops
- In effect, each write creates a new variable, written only once
- Hardware implications
  - with multiple instructions being executed, a dead variable can be inferred on each write (previous instructions may still need to read it)
  - A different buffer can be assigned the new value while the old value is still live!

## The Stack

- Modern programming languages require the ability to allocate space for an indefinite number of variables
- Each instance of a method requires its own space for variables, arguments, and temps.
- The *Stack of Activation Records* is a data structure that satisfies this requirement.
  - On invocation
    - Allocate space for arguments, temps, local variables: a *Frame*
    - Save (spill) some registers to allocate for subroutine
    - Save linkage information (how to return)
    - Transfer control to subroutine
  - On return
    - Assign return value
    - Restore spilled registers
    - Deallocate space
    - Jump back to original code

## Caller vs. Callee

- Who should allocate space?
  - Callee knows how much space it needs
  - Arguments and return are special: they are shared
- Who should save registers?
- Caller should save
  - Don't need to save registers not being used
  - Only caller knows this
- Callee should save
  - Don't need to save registers that won't be touched
  - Only callee knows
- Solution: do both!

2-May-07

CS210

89

## Recommended Readings

For today's lecture

- Chapter 11: function invocation.
- Randy Bryant, *Alpha Assembly Language Guide* (available under Resources at the website) Section 3
- Chapter 12: assembling and disassembling

For the mini-assignment

- Chapter 6: program structure
- Chapter 7: strings
- Chapter 8: running the simulator
- Chapter 10: writing and debugging in assembly language

2-May-07

CS210

90

## Caller/Callee Register Allocation

- Temporary registers for callee
  - \$t0-\$t11
  - Free for use, but not preserved
- Saved registers for caller
  - \$s0-\$s5
  - Free to use, but responsible for saving/restoring value
- Every method is potentially both a caller and callee
  - Leaves (methods that invoke no other methods) often don't need to use S registers—no spills
  - Other nodes save registers they use exactly once: on invocation

2-May-07

CS210

91

## Dealing with Arguments

- Used for communication between caller and callee
- No limit to number of allowed arguments
  - Pass arguments in registers: \$a0-\$a5
  - Pass additional arguments through stack
- Argument registers \$a0-\$a5 are like temporaries
  - Must be preserved if needed after a call
  - If not needed, can be used as a temporary

2-May-07

CS210

92

## Use of Stack for Subroutines: Caller

- Caller has allocated space for arguments beyond \$a4 in its stack frame
  - Save current (caller's) arguments on stack if needed
  - Save previously returned result \$v0 (if needed)
- Assign arguments to registers (\$a0-\$a5)
- If temporary registers are live, save
- Caller executes bsr instruction
  - Address of subsequent instruction stored in \$ra
  - Jumps to beginning of callee
- On return
  - Restore arguments (\$a0-\$a5) and tmps (\$t0-\$t5) if/when needed

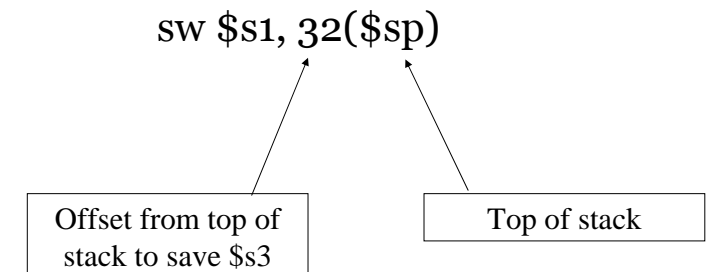
## Use of Stack for Subroutines: Callee

- Allocate space for new activation record
- Save any saved registers (\$s0-\$s11) to stack
- Save \$ra to stack if any other procedure might be called
- Perform function (possibly invoking other functions)
- Restore saved registers (\$s0-\$s11, \$ra)
- Assign return value to \$v0
- Deallocate space for current activation record
- Return to calling procedure via \$ra

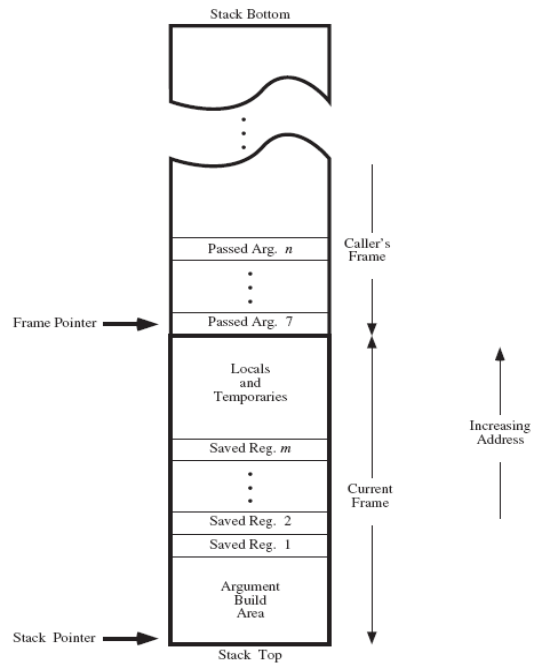
## Accesses to the Stack

- The layout of a stack frame (activation record) is determined when the method is *compiled*
- At assembly time, when the code is produced
  - the absolute address cannot be fixed (it varies depending on circumstances)
  - the relative address (relative to the top of stack) is known: a small constant
- Addressing mode of base register + displacement is perfect
  - base: frame pointer (or stack pointer)
  - displacement (computed when the stack frame is laid out.

## Example of Stack Access







2-May-07

97

## Optimizations

- Stack is designed to handle worst case:
  - Spilled registers
  - Return address
  - Extra arguments
- In practice stack can be very small
  - If called procedure is a leaf (does not call other procedures), it may not need a stack at all.
  - Even if it calls other procedures, it needs to save RA, but
    - May not need to save arguments
    - May not need to save registers
    - Could be as little as one word!

2-May-07

CS210

98

Computer Science 210  
**Computer Systems 1**  
 2007 Semester 1  
**Lecture Notes Part 2**  
**The Assembly Process**

*James Goodman*



2-May-07

CS210

100

## Recommended Readings

- Hutton's Notes, Chapter 12: assembling and disassembling

## Translation vs. Interpretation

- A program written in language L defines a “machine”
- Problem:
  - We have a program written in language L1
  - We have a computer that understands how to execute language L2
  - How to “execute” program?
- Solution 1: Compilation/Translation/Assembly
  - A “compiler” takes as input a program written in L1 and creates a program written in L2
- Solution 2: Interpretation/Simulation/Emulation
  - A “program” takes as input a program written in L1 and walks through its execution, taking input and creating output as if it were an L1 computer.

2-May-07

CS210

101

## The assembly process: Overview

- A computer understands machine code
- People (and compilers) write assembly language
  - Today it’s usually compilers
- Goal: create a file describing exactly what memory should look like before starting execution of a program
- Assembly: the process of translating a program written in assembly language into a program written in machine language.
  - Machine language is specific to a computer
  - Need to create memory image that includes
    - instructions (including links to libraries, kernel, etc.)
    - data (constants, static variables, dynamic variables)

2-May-07

CS210

102

## Steps in Assembly

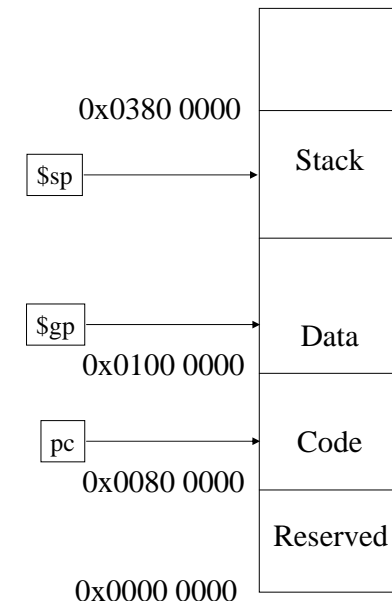
- Pass 1: Scan program and parse
  - Identify declarations of instructions and static data
  - identify pseudo-instructions
  - Allocate space for instructions and data
  - Recognize labels
    - Define address if possible
    - Remember for future reference: Symbol Table
  - For data
    - Allocate space
    - Associate label with address
    - Generate appropriate representation
- Pass 2: Rescan and generate code
  - Translate instruction to machine code

2-May-07

CS210

103

## A Picture of Memory



2-May-07

CS210

104

# Program to Assemble

```
data {
  number:
    quad      32769;
  string1:
    asciiz: "Hello!\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;
    clr      $a1;
    ldiq     $a0, 0;          / CALLSYS_EXIT;
    call_pal 0x83;           / CALL_PAL_CALLSYS;
} code
```

# ASCII Characters

H	0x48
e	0x65
l	0x6c
l	0x6c
o	0x6f
!	0x21
\n	0x0a
0	ox00

# Symbol Table

Symbol	Address
number	0x0100 0000
string1	0x0100 0004
xxx	0x0080 0004
yyy	0x0080 0010

# Instruction Formats

Figure 1-1: Instruction Format Overview

	31	26	25	21	20	16	15		5	4	0	
Opocode	Number											PALcode Format
Opocode	RA	Disp										Branch Format
Opocode	RA	RB	Disp									Memory Format
Opocode	RA	RB	Function					RC				Operate Format

– From The Alpha Architecture Handbook, Compaq Computer Corporation, 1998.

## Opcode Assignment

Instruction	Format	Opcode	Function code
beq	Branch	0x39	
bne	Branch	0x3d	
br	Branch	0x30	
bis	Operate	0x11	0x20
ldq	Memory	0x29	
addq	Operate	0x10	0x20
subq	Operate	0x10	0x29
cal_pal	PALcode	0x00	

2-May-07

CS210

109

## Register Names

\$0	\$v0
\$1-\$8,	\$t0-\$t9
\$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 (special)

2-May-07

CS210

110

## Program to Assemble

```

data {
    number:
        quad    32769;
    string1:
        asciiz: "Hello!\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;
        clr     $a1;
        ldiq    $a0, 0;          / CALLSYS_EXIT;
        call_pal 0x83;          / CALL_PAL_CALLSYS;
} code

```

2-May-07

CS210

111

## Working Assembly

```

                                0x32    $t0=1                +3
0x0080 0000 1100 10|00 001|0 0000 0000 0000 0000 0011 0000
                                0x29    $s0=9    $t0=1        +0000
0x0080 0004 1010 01|01 001|0 0001|0000 0000 0000 0000 0000
0x0080 0008
...

0x0100 0000 0000|0000|0000|0000|1000|0000|0000|0001
0x0100 0004 0000 0000 0000 0000 0000 0000 0000 0000
0x0100 0008 0110 1100|0110 1100|0110 0101|0100 1000
0x0100 000c 0000 0000|0000 1010|0010 0001|0110 1111
0x0100 0010
...

```

2-May-07

CS210

112

# Assembled Code

```
00800000 e4200003 beq $t0, .main.yyy
00800004 a5210000 ldq $s0, +0000($t0)
00800008 41203529 subq $s0, 01, $s0
0080000c f53ffffd bne $s0, .main.xxx
00800010 43ef7401 addq $zero, 7b, $t0
00800014 c3fffffb br $zero, main.xxx
00800018 47ff0411 bis $zero, $zero, $a1
0080001c a61d0000 ldq $a0, +0000($gp)
00800020 00000083 call_pal 0000083
00800024 00000000 call_pal 0000000
```

```
.main.number:
01000000 00008001 Hex 8001
01000004 00000000

.main.string1:
01000008 6c6c6548 Hell Hex a216f6c6c6548
0100000c 000a216f o!??
```