Computer Science 210

Computer Systems 1

2007 Semester 1

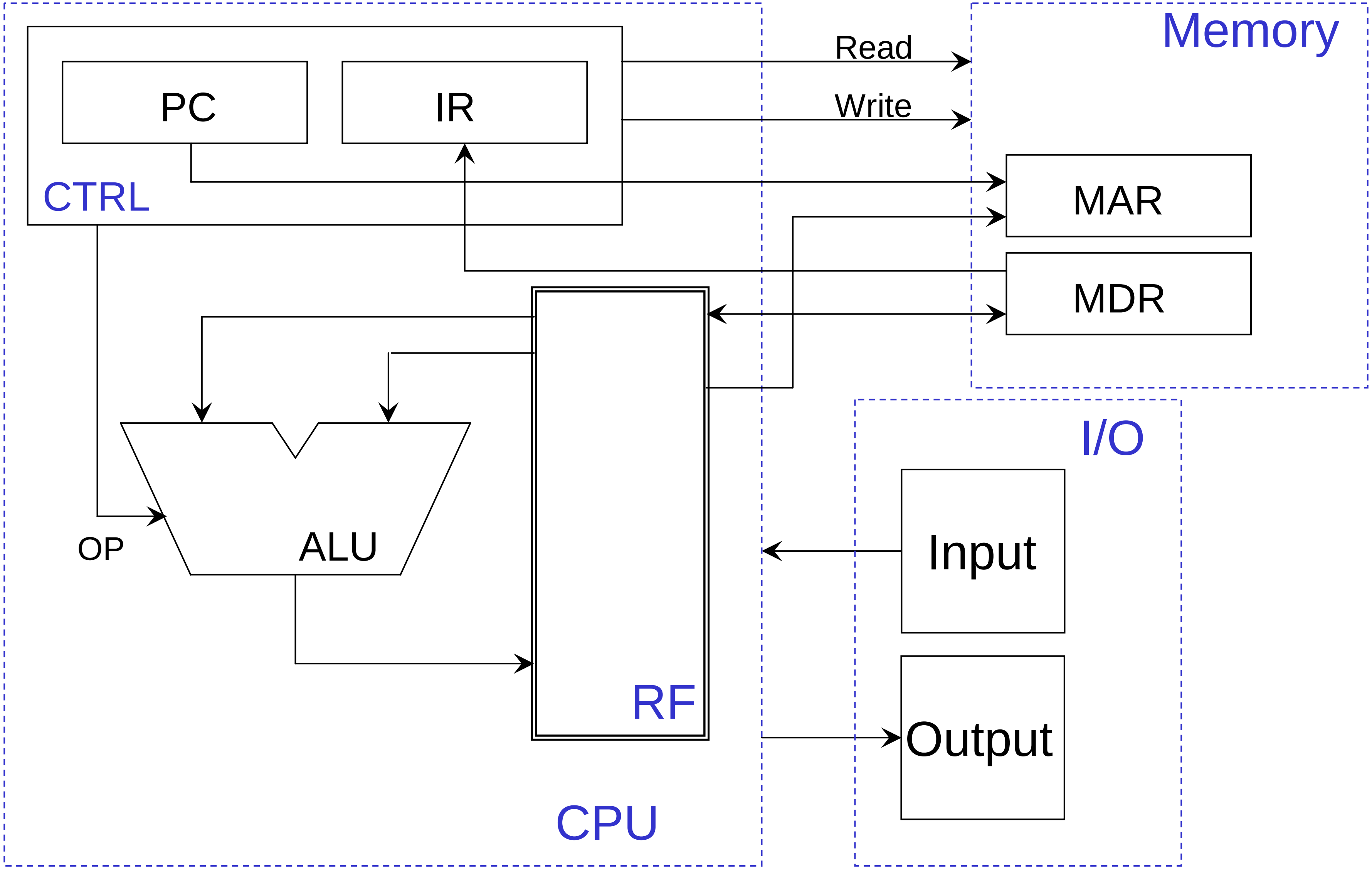
Tutorial

The Programmer's View of Computer Hardware

James Goodman

Department of Computer Science

The Alpha Computer



Four Categories of Instructions

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

Registers

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

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)

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

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

Control Instructions

Basic instruction for choosing alternate instruction path:

- Branch on condition (Kiwi-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

Which is Correct View of Memory?

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

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

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

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

Byte Order

- Little Endian

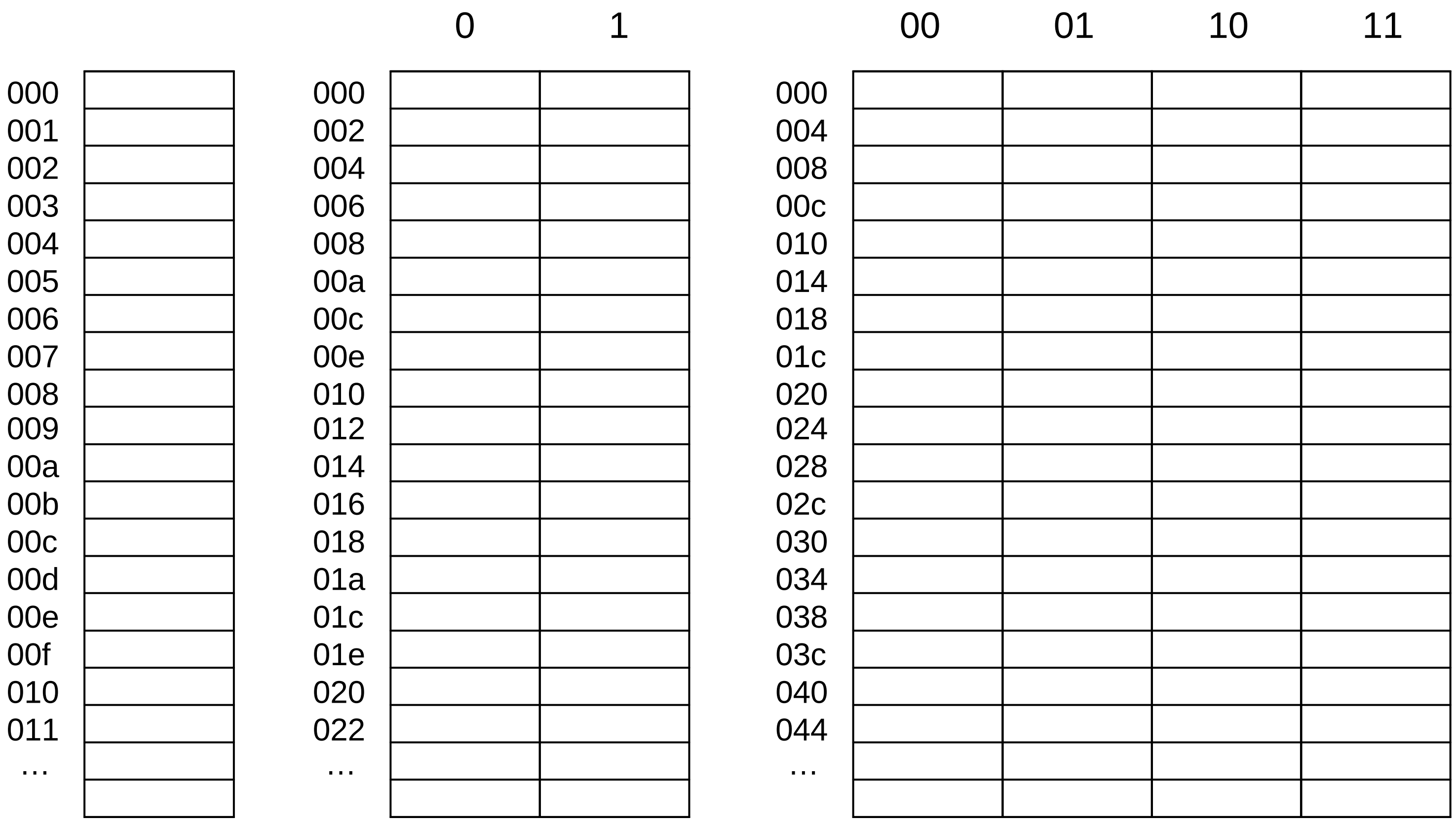
000	
001	
002	
003	
004	
005	
006	
007	
008	
009	
00a	
00b	
00c	
00d	
00e	
00f	
010	
011	
…	

	1	0
000		
002		
004		
006		
008		
00a		
00c		
00e		
010		
012		
014		
016		
018		
01a		
01c		
01e		
020		
022		
…		

	11	10	01	00
000				
004				
008				
00c				
010				
014				
018				
01c				
020				
024				
028				
02c				
030				
034				
038				
03c				
040				
044				
…				

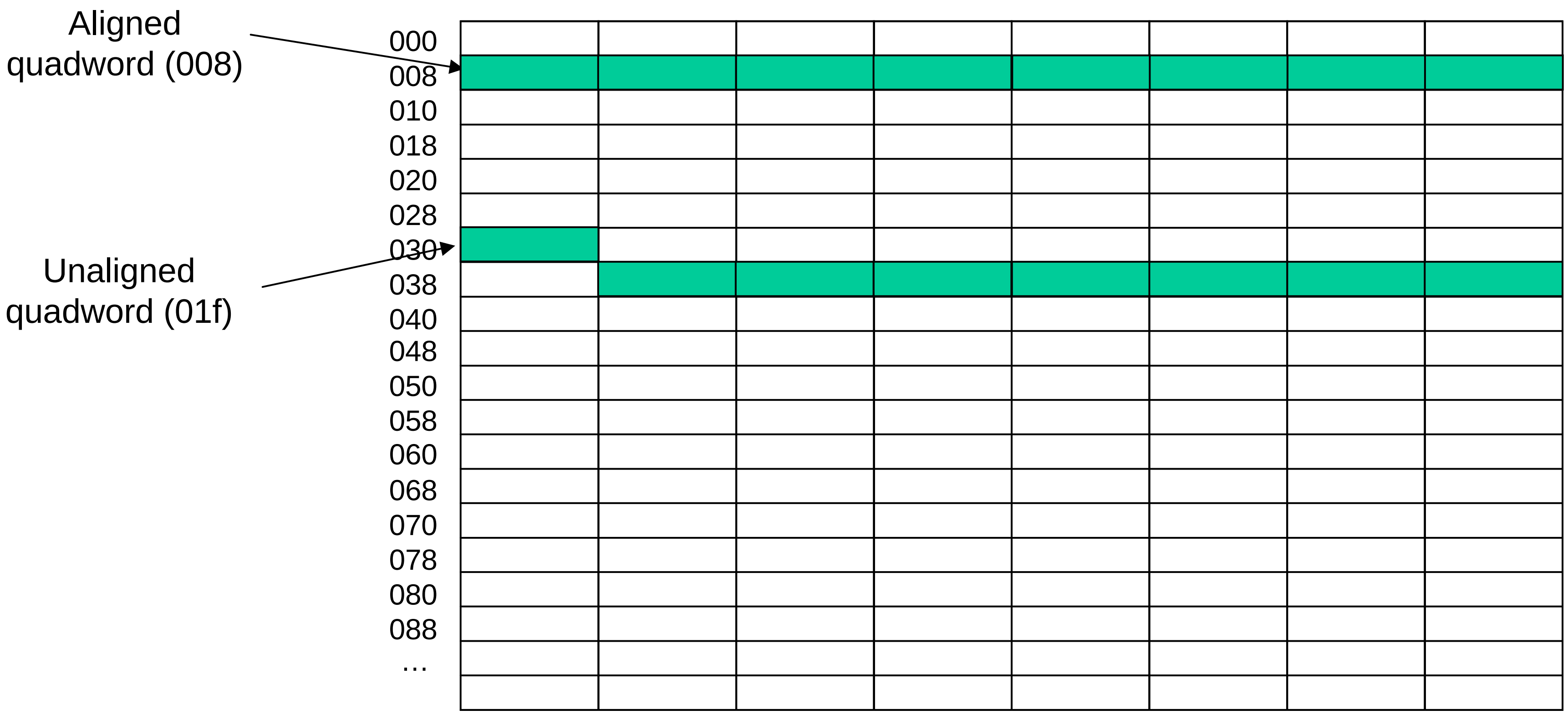
Byte Order

- Big Endian



Views of Memory

- An array of quadwords (little-endian)



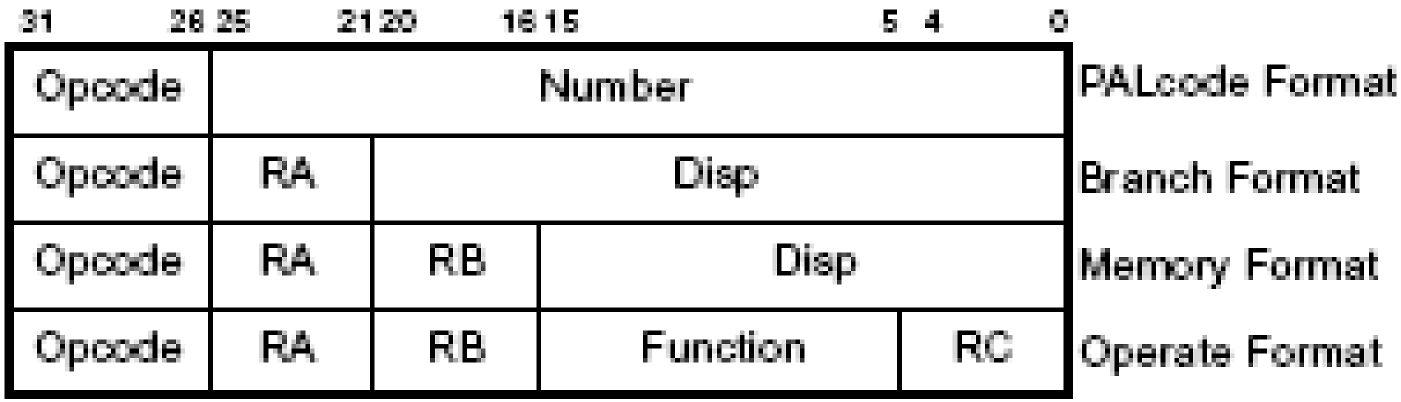
Load Instructions

- **`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

Motivation: Instruction Formats

Figure 1–1: Instruction Format Overview



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

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

11 1010	Reg	Target
6 bits	5 bits	21 bits ???

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

Branch Instruction

- How to specify full add (≤ 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 * Target)

Long-distance Branches

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

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!

Optimization: lda, ldah

lda	Reg A	Reg B	<low address>
6 bits	5 bits	5 bits	16 bits

ldah	Reg A	Reg B	<hi address>
6 bits	5 bits	5 bits	16 bits

- **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

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

Load Reg, Disp(Base)

Idq	Dest	Base	Displacement
6 bits	5 bits	5 bits	16 bits

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

“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
```

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)

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.

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

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)

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!

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

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

Accesses to the Stack

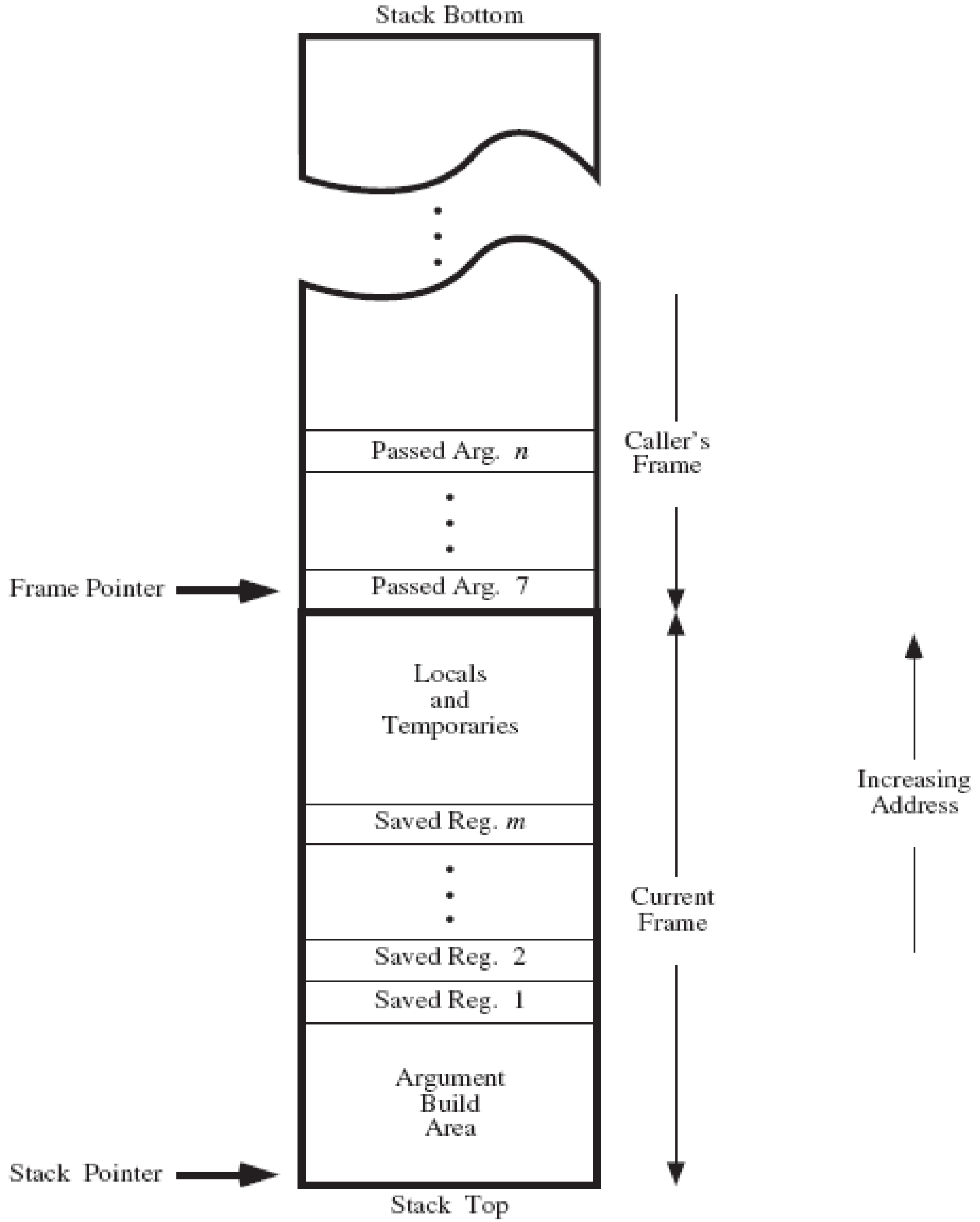
- The layout of a stack frame (activation record) is determined when the method is *compiled*
- At assembly time, when the code is produced
  - the abolute address cannot be fixed (it varies depending on circumstances)
  - the relative address (relative to the top of stack) is known: a small constant
- 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

sw $s1, 32($sp)

Offset from top of stack to save $s3

Top of stack



Stack Bottom
Passed Arg. n
Passed Arg. 7
Caller's Frame
Frame Pointer
Locals and Temporaries
Saved Reg. m
Saved Reg. 2
Saved Reg. 1
Argument Build Area
Current Frame
Increasing Address
Stack Pointer
Stack Top

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 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!

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.

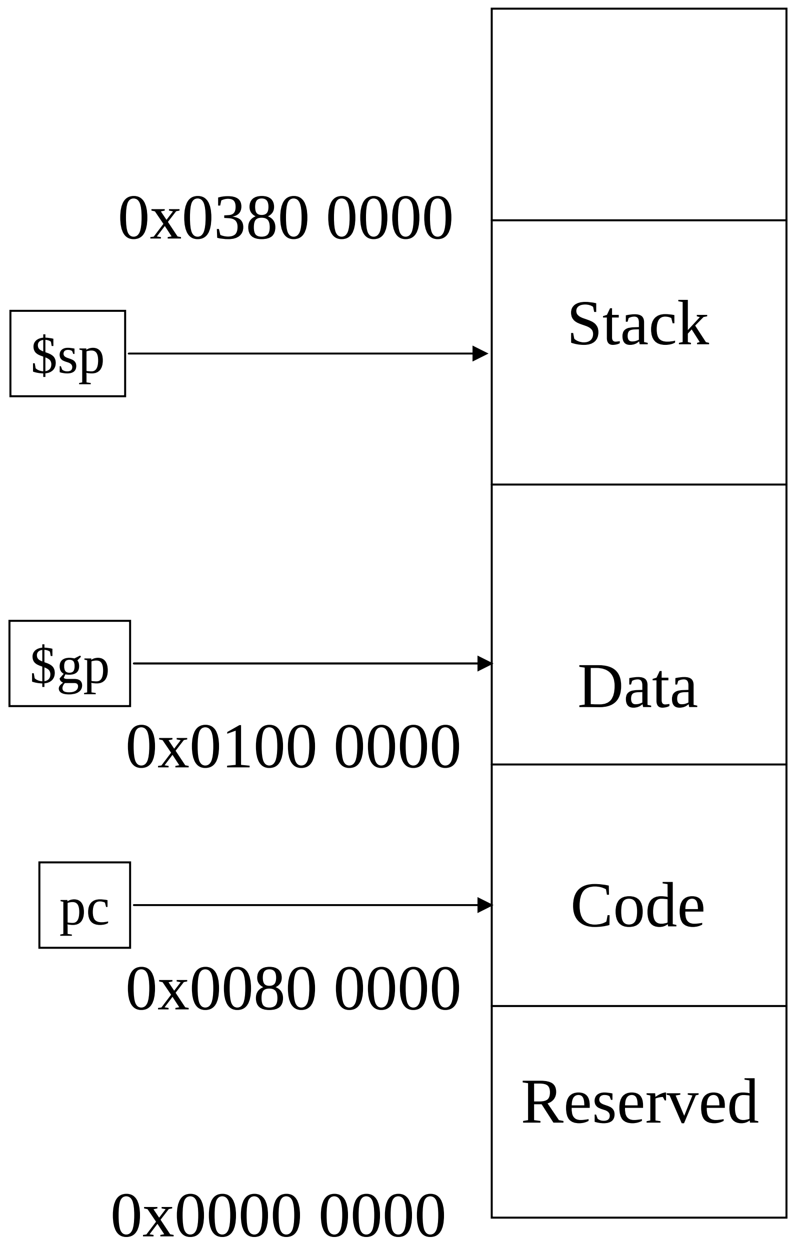
The assembly process: Overview

- 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

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

A Picture of Memory



Symbol Table

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

Floating Point in the Alpha

- The Alpha has 32 floating-point registers \$f0-\$f31
  - \$f31 is always zero
- Instructions are same format as integer instructions
  - Floating-point registers are implied (mostly)
  - No literals allowed
- Two levels of precision
  - S: "single-precision" 32-bit IEEE format
  - T: "double-precision" 64-bit IEEE format

Arithmetic Instructions

- Arithmetic instructions have separate “operate” opcode
- Instructions are either single (S) or double (T) type
- Operations
  - ADD
  - SUB
  - MUL
  - DIV
  - SQRT

Load/Store Instructions

- Load: LDS, LDT
  - Address comes from integer register
  - Same as ldl, ldq but target is FP register
- Store: STS, STT
  - Address comes from integer register
  - Same as stl, stq but source is FP register

Control Instructions

- Branch on condition
  - Same as integer, but using FP reg (6 cases)
  - Additional problem: NAN
- Compare two FP regs
  - EQ, GE, GT, LE, LT, NE
  - Result set zero/non-zero value in FP reg
  - Also “unordered”
- Conditional move instructions
  - Test FP register *a* against zero: EQ, GE, GT, LE, LT, NE
  - If true, copy register *b* into register *c*

Other Instructions

- Move: copy between I register and FP register
  - FtoIS,FtoIT,ItoFS,ItoFT
  - No format change
- Convert between floating point/Integer
  - convert in-place in FP registers
  - S <=> T; T <=> Q ; Q => S (no S=> Q)
  - Longword <=> Quadword
- CPYS: Copy sign bit to destination (merge)
- CPYSN: Copy and invert sign bit to destination
- CPYSE: Copy sign and exponent to destination

Computer Science 210

Computer Systems 1

2007 Semester 1

Lecture Notes Part 2

2. Performance: the Big Picture

Lecture 12
18 May 07

James Goodman

Department
of
Computer Science

Performance

Question:

"How long does it take to execute an instruction?"

Answer:

"It depends"

Modern Processors

Are pipelined (Assembly-line process)

One stage per clock cycle

- Stage 1: Fetch instruction
- Stage 2: Decode instruction
- Stage 3: Fetch operands
- Stage 4: Perform operation
- Stage 5: Put away result

Problems

- Need to fetch multiple operands in stage 3
- Sometimes operand is not ready
  - Producing instruction has not completed
  - Must *stall* pipeline
- For load instruction, operation is to memory
  - Memory is slow
- What happens on a conditional branch???

Summary

Performance optimized by making the common case fast

- The uncommon case merely must execute correctly
- This approach has been highly successful, but results in large variance in execution time

Memory plays a critical role in performance

- Many programs spend *more time waiting for memory than executing instructions*

Implications for programmers

- Placement of data is critical
  - Temporal & spatial locality can be exploited
- Branches are expensive
  - Unpredictable branches are especially difficult