CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Instructors:
Randy H. Katz
David A. Patterson

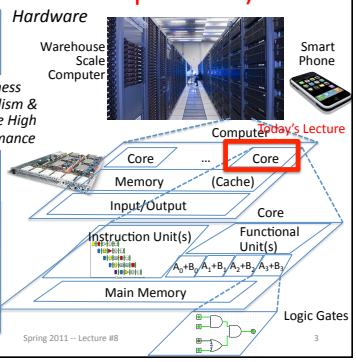
http://inst.eecs.Berkeley.edu/~cs61c/sp11

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

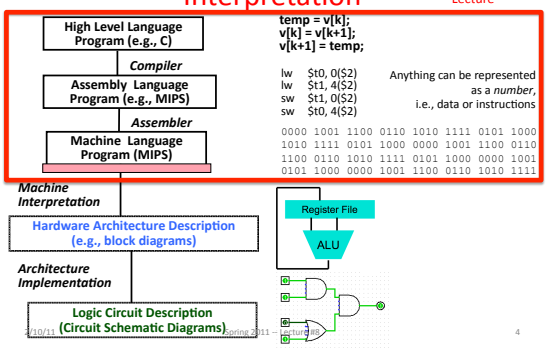
- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time

Harness Parallelism & Achieve High Performance



Levels of Representation/ Interpretation

Today's Lecture



Agenda

- Review
- Instructions as Numbers
- Administrivia
  – Secret to Getting a Good Job / Good Internship
- Assemblers
- Compilers and Linkers
- Technology Break
- Compilers vs. Interpreters
- Compiler Optimization?

Review

- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, …
- Integers have largest positive and largest negative numbers, but represent all in between
  – Two's comp. weirdness is one extra negative numInteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Floating point is an *approximation* of reals
  – 2^{32} patterns to represent reals from $-\infty$ to $+\infty$
- Everything is a (binary) number in a computer

Correction

- A = (1000000.0 + 0.000001) - 1000000.0
- B = (1000000.0 - 1000000.0) + 0.000001
- In single precision floating point arithmetic, A does not equal B
  A = 0.000000, B = 0.000001
- Floating Point Addition is not Associative!
  – Integer addition is associative
- When does this matter?

Instructions as Numbers

- Instructions are also kept as binary numbers in memory
  – Stored program concept
  – As easy to change programs as it is to change data
- Register names mapped to numbers
- Need to map instruction operation to a part of number

Instructions as Numbers

- `addu $t0,$s1,$s2`
  – Destination register `$t0` is register 8
  – Source register `$s1` is register 17
  – Source register `$s2` is register 18
  – Add unsigned instruction encoded as number 33

0	17	18	8	0	33
000000	10001	10010	01000	00000	100001
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Groups of bits call *fields* (unused field default is 0)
- Layout called *instruction format*
- Binary version called *machine instruction*

Instructions as Numbers

- `sll $zero,$zero,0`
  – `$zero` is register 0
  – Shift amount 0 is 0
  – Shift left logical instruction encoded as number 0

0	0	0	0	0	0
000000	00000	00000	00000	00000	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Can also represent machine code as base 16 or base 8 number: $0000\ 0000_{hex}$, 0000000000_{oct}

Everything in a Computer is Just a Binary Number

- Up to program to decide what data means
- Example 32-bit data shown as binary number:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$

What does it mean if its treated as

1. Signed integer
2. Unsigned integer
3. Floating point
4. ASCII characters
5. Unicode characters
6. *MIPS instruction*

Student Roulette?

Implications of Everything is a Number

- *Stored program concept*
  – Invented about 1947 (many claim invention)
- As easy to change programs as to change data!
- Implications?

Student Roulette?

Names of MIPS fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand.
- *rd*: register destination operand (result of operation)
- *shamt*: Shift amount.
- *funct*: Function. This field, often called *function code*, selects the specific variant of the operation in the op field

What about Load, Store, Immediate, Branches, Jumps?

- Fields for constants only 5 bits (-16 to +15)
  - Too small for many common cases
- #1 Simplicity favors regularity (all instructions use one format) vs. #3 Make common case fast (multiple instruction formats)?
- 4th Design Principle: *Good design demands good compromises*
- Better to have multiple instruction formats and keep all MIPS instructions same *size*
  - All MIPS instructions are 32 bits or 4 bytes

Names of MIPS Fields in I-type

op	rs	rt	address or constant
6 bits	5 bits	5 bits	16 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand for branches but register destination operand for lw, sw, and immediate operations
- *Address/constant*: 16-bit two's complement number
  - Note: equal in size of rd, shamt, funct fields

Register (R), Immediate (I), Jump (J) Instruction Formats

	op	rs	rt	rd	shamt	funct
R-type	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Now loads, stores, branches, and immediates can have 16-bit two's complement address or constant: -32,768 (-2^{15}) to +32,767 ($2^{15}-1$)
- What about jump, jump and link?

	op	address
J-type	6 bits	26 bits

Addressing in Branches

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Programs much bigger than 2^{16} bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: *PC-relative branching*

Addressing in Branches

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Hardware increments PC early, so relative address is PC = (PC + 4) + Branch address
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  PC = (PC + 4) + (Branch address*4)

Addressing in Jumps

J-type	op	address
	6 bits	26 bits

- Same trick for Jumps, Jump and Link
  PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Encoding of MIPS Instructions: Must Be Unique!

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
subu	R	0	reg	reg	reg	0	35_{ten}	n.a.
sltu	R	0	reg	reg	reg	0	43_{ten}	n.a.
sll	R	0	n.a.	reg	reg	constant	0_{ten}	n.a.
addi unsigned	I	9_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address
beq	I	4_{ten}	reg	reg	n.a.	n.a.	n.a.	address
bne	I	5_{ten}	reg	reg	n.a.	n.a.	n.a.	address
j (jump)	J	2_{ten}	n.a.	n.a.	n.a.	n.a.	n.a.	address
jal	J	3_{ten}	n.a.	n.a.	n.a.	n.a.	n.a.	address
jr (jump reg)	R	0	reg	reg	reg	0	8_{ten}	n.a.

Agenda

- Review
- Instructions as Numbers
- Administrivia
  - Secret to Getting a Good Job / Good Internship
- Assemblers
- Compilers and Linkers
- Technology Break
- Compilers vs. Interpreters
- Compiler Optimization?

Secret To Getting Good Job/Internship

- Job/Intern interviews are (now) oral exams
- Long-term memory vs. short-term/working memory
  "Long-term memory is intended for storage of information over a long time. Information from the working memory is transferred to it after a few seconds. Unlike in working memory, there is little decay."
- Learning for recall 6-`1 months later in oral exam vs. getting grades?
  - Read before lecture, think about lecture, do assignments, labs , ..?
  - Cram day before exam, start project with 24 hours to go?

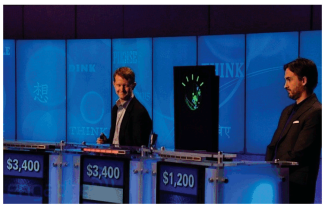
Scores on Project 1, Part 1

Number grades:	324
Mean:	22.0
Mode:	25.0
Standard deviation:	5.5
Minimum:	0.0
1st quartile:	20.0
2nd quartile (median):	25.0
3rd quartile:	25.0
Maximum:	25.0

Distribution:

0.0 - 2.5:	8	*
2.5 - 5.0:	1	*
5.0 - 7.5:	2	*
7.5 - 10.0:	10	*
10.0 - 12.5:	3	*
12.5 - 15.0:	4	*
15.0 - 17.5:	14	**
17.5 - 20.0:	5	*
20.0 - 22.5:	56	***…*
22.5 - 25.0:	4	*
25.0 - 27.5:	217	*****…*

Computers In The News



- IBM Watson play Jeopardy! with champions
  - A significant milestone in computing, on par with IBM Deep Blue vs. Kasparov in chess in 1997?
- Mon 2/14 – Wed 2/16 7-7:30PM KGO Channel 7

http://www-03.ibm.com/innovation/us/watson

My Life Beyond Computing

- 1935 Ford Station Wagon
- 221 cubic inch (85 HP) original Ford Flathead V8 Engine

My Life Beyond Computing

- 1938 Ford Cabriolet
- 454 cubic inch (7500 cc) "Big Block" Chevy V8 Engine

Assembler

- Input: Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1) _

addu $t0,$s2,$t0 _

sw $t0,1200($t1) _

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1) _I

addu $t0,$s2,$t0 _

sw $t0,1200($t1) _

		9	8	1200		

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1) I_

addu $t0,$s2,$t0 R_

sw $t0,1200($t1) _

		9	8	1200		
		18	8	8	0	

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1)	_		9	8	1200		
addu $t0,$s2,$t0	_		18	8	8	0	
sw $t0,1200($t1)	_		9	8	1200		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1)	_		9	8	1200		
addu $t0,$s2,$t0	_		18	8	8	0	
sw $t0,1200($t1)	_		9	8	1200		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

&A=$t1 (reg 9), $t0 (reg 8), h=$s2 (reg 18)

A[300] = h + A[300];

Student Roulette?

Format?

lw $t0,1200($t1)	_		9	8	1200		
addu $t0,$s2,$t0	_		18	8	0	0	
sw $t0,1200($t1)	_		9	8	1200		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting to MIPS Machine code

Address	Loop:	Format?						
800	sll $t1,$s3,2	_						
804	addu $t1,$t1,$s6	_						
808	lw $t0,0($t1)	_						
812	bne $t0,$s5, Exit	_						
816	addiu $s3,$s3,1	_						
820	j Loop	_						
	Exit:							

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting to MIPS Machine code

Address	Loop:	Format?						
800	sll $t1,$s3,2	R	0	0	19	9	2	0
804	addu $t1,$t1,$s6	R	0	9	22	9	0	33
808	lw $t0,0($t1)	I	35	9	8	0		
812	bne $t0,$s5, Exit	I	5	8	21	2		
816	addiu $s3,$s3,1	I	8	19	19	1		
820	j Loop	J	2	200				
	Exit:							

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

32 bit constants in MIPS

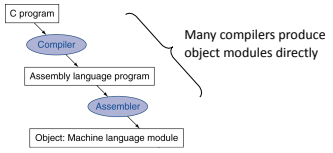
- Can create a 32-bit constant from two 32-bit MIPS instructions
- *Load Upper Immediate (*`lui` *or* "Louie") puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0? $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{two}$

lui $s0, 61 # 61 = $0000\ 0000\ 0011\ 1101_{two}$

ori $s0, $s0, 2304 # 2304 = $0000\ 1001\ 0000\ 0000_{two}$

Translation

§2.12 Translating and Starting a Program



Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called *assembly,* program called *assembler*
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called *assembly language*
- Can also accept instructions convenient for programmer but not in hardware
  - *Load immediate* (*li*) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - *Load double* (*ld*) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - Called *Pseudo-Instructions*

Assembler Directives (p. B-5 to B-7)

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.asciiz str`: Store the string `str` in memory and null-terminate it
  - `.word w1…wn`: Store the *n* 32-bit quantities in successive memory words

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1     →  add $t0, $zero, $t1
blt $t0, $t1, L   →  slt $at, $t0, $t1
                     bne $at, $zero, L
```

  - $at (register 1): assembler temporary

More Pseudoinstructions

Student Roulette?

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`addu $t0,$t6,1`	_____
`subu $sp,$sp,32`	_____
`sd $a0, 32($sp)`	_____
	_____
`la $a0, str`	_____

More Pseudoinstructions

Student Roulette?

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`subu $sp,$sp,32`	_____
`sd $a0, 32($sp)`	_____
	_____
`la $a0, str`	_____

More Pseudoinstructions

Student Roulette?

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	_____
	_____
`la $a0, str`	_____

More Pseudoinstructions

Student Roulette?

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)`
	`sw $a1, 36($sp)`
`la $a0, str`	`lui $at,left(str)`
	`ori $a0,$at,right(str)`

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Separate Compilation and Assembly

- No need to compile all code at once
- How put pieces together?

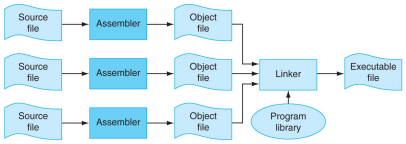
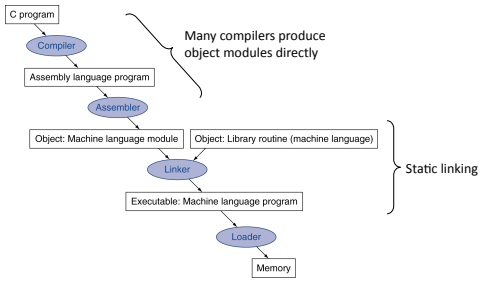


FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.

Translation and Startup

§2.12 Translating and Starting a Program



Linker Stitches Files Together

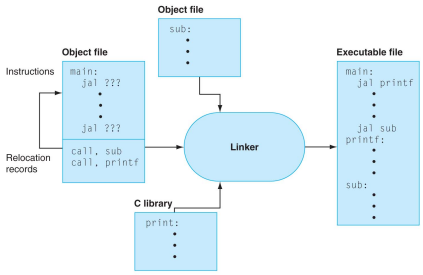


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Often a slower than compiling
  - all the machine code files must be read into memory and linked together

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (cover later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do "exit" systems call

Review

- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- Assemblers can enhance machine instruction set to help assembly-language programmer
- Translate from text that easy for programmers to understand into code that machine executes efficiently: Compilers, Assemblers
- Linkers allow separate translation of modules
- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer