CS 61C: Great Ideas in Computer Architecture

MIPS Instruction Representation II

Guest Lecturer: Justin Hsia

Review of Last Lecture

- **Simplifying MIPS:** Define instructions to be same size as data word (one word) so that they can use the same memory
  - Computer actually stores programs as a series of these 32-bit numbers
- **MIPS Machine Language Instruction:** 32 bits representing a single instruction

R:	`opcode`	`rs`	`rt`	`rd`	`shamt`	`funct`
I:	`opcode`	`rs`	`rt`	`immediate`		

Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g. MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

We are here

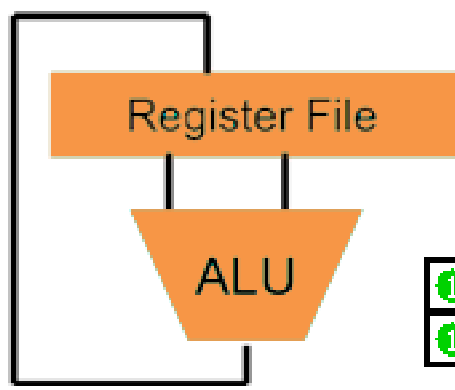
Assembler

Machine Language Program (MIPS)

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

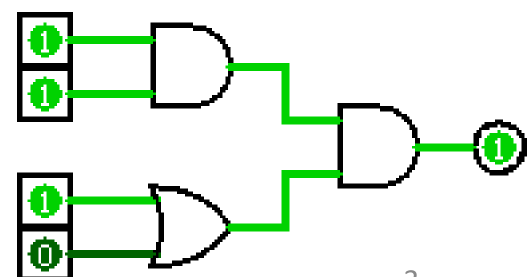
Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

I-Format Immediates

- `immediate` (16): *two's complement* number
  - All computations done in words, so 16-bit immediate must be *extended* to 32 bits
  - Green Sheet specifies ZeroExtImm or SignExtImm based on instruction
- Can represent 2^{16} different immediates
  - This is large enough to handle the offset in a typical `lw`/`sw`, plus the vast majority of values for `slti`

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Sometimes want to use immediates > ± 2^{15} with `addi`, `lw`, `sw` and `slti`
  - Bitwise logic operations with 32-bit immediates
- **Solution:** Don't mess with instruction formats, just add a new instruction
- Load Upper Immediate (`lui`)
  - `lui reg,imm`
  - Moves 16-bit `imm` into upper half (bits 16-31) of `reg` and zeros the lower half (bits 0-15)

`lui` Example

- Want: `addiu $t0,$t0,0xABABCDCD`
  - This is a pseudo-instruction!
- Translates into:

```
lui   $at,0xABAB          # upper 16
ori   $at,$at,0xCDCD      # lower 16
addu  $t0,$t0,$at         # move
```

Only the assembler gets to use $at

- Now we can handle everything with a 16-bit immediate!

Branching Instructions

- `beq` and `bne`
  - Need to specify an address to go to
  - Also take two registers to compare
- Use I-Format:

31			0
opcode	rs	rt	immediate

  - `opcode` specifies `beq` (4) vs. `bne` (5)
  - `rs` and `rt` specify registers
  - How to best use `immediate` to specify addresses?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing: Use the `immediate` field as a two's complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify $\pm 2^{15}$ addresses from the PC
- So just how much of memory can we reach?

Branching Reach

- **Recall:** MIPS uses 32-bit addresses
  - Memory is byte-addressed
- Instructions are *word-aligned*
  - Address is always multiple of 4 (in bytes), meaning it ends with `0b00` in binary
  - Number of bytes to add to the PC will always be a multiple of 4
- Immediate specifies words instead of bytes
  - Can now branch $\pm 2^{15}$ words
  - We can reach 2^{16} instructions = 2^{18} bytes around PC

Branch Calculation

- If we don't take the branch:
  - `PC = PC + 4` = next instruction
- If we do take the branch:
  - `PC = (PC+4) + (immediate*4)`
- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)
  - Branch from PC+4 for hardware reasons; will be clear why later in the course

Branch Example (1/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1     1
      j     Loop         2
End:                     3
```

Start counting from instruction AFTER the branch

- I-Format fields:

  `opcode` = 4 (look up on Green Sheet)

  `rs` = 9 (first operand)

  `rt` = 0 (second operand)

  `immediate` = 3

Branch Example (2/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Field representation (decimal):

31 … 0

4	9	0	3

Field representation (binary):

31 … 0

000100	01001	00000	0000000000000011

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no
- What do we do if destination is > 2^{15} instructions away from branch?
  - Other instructions save us
  - beq $s0,$0,far → bne $s0,$0,next / j far / next: # next instr

```
beq $s0,$0,far            bne $s0,$0,next
# next instr    -->       j   far
                    next: # next instr
```

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Administrivia

- Project 1 Part 1 due Sunday 2/17
  - No homework! (still have discussion and lab)
  - Solo project
- Midterm in 3 weeks (3/4)
  - Never too early to start looking at past exams
  - Midterm Review: Fri 3/1, 6-9pm, 2050 VLSB (this room!)

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

J-Format Instructions (1/4)

- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 6-bit opcode and a 32-bit address into a single 32-bit word

J-Format Instructions (2/4)

- Define two "fields" of these bit widths:

31	0
6	26

- As usual, each field has a name:

31	0
opcode	target address

- **Key Concepts:**
  - Keep opcode field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address

J-Format Instructions (3/4)

- We can specify 2^{26} addresses
  - Still going to word-aligned instructions, so add `0b00` as last two bits (multiply by 4)
  - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate almost all of the time, since programs aren't that long
  - Only problematic if code straddles a 256MB boundary
- If necessary, use 2 jumps or `jr` (R-Format) instead

J-Format Instructions (4/4)

- Jump instruction:
  - New PC = { (PC+4)[31..28], target address, 00 }
- Notes:
  - { , , } means concatenation
    { 4 bits , 26 bits , 2 bits } = 32 bit address
    - Book uses || instead
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC

Question: When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) Jump instructions don't require any changes
2) Branch instructions don't require any changes

	1	2
a)	F	F
b)	F	T
c)	T	F
d)	T	T

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via addition with `0`
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

`move dst,src` translated into
`addi dst,src,0`

Assembler Pseudo-Instructions

- List of pseudo-instructions: http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
  - List also includes instruction translation
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
  - Reserve a register ($1 or $at for "assembler temporary") that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it's not safe to code with it

MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each MAL instruction becomes 1 or more TAL instruction
- TAL $\subset$ MAL

Summary

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
  - Branches use PC-relative addressing

I:	opcode	rs	rt	immediate

- J-Format: `j` and `jal` (but not `jr`)
  - Jumps use absolute addressing

J:	opcode	target address

- R-Format: all other instructions

R:	opcode	rs	rt	rd	shamt	funct

BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

They have been prepared in a way that should be easily readable.

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Assembly Practice

- Assembly is the process of converting assembly instructions into machine code
- On the following slides, there are 6-lines of assembly code, along with space for the machine code
- For each instruction,
  1) Identify the instruction type (R/I/J)
  2) Break the space into the proper fields
  3) Write field values in decimal
  4) Convert fields to binary
  5) Write out the machine code in hex
- Use your Green Sheet; answers follow

Code Questions

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2

 804  addu  $t1,$t1,$s6

 808  lw    $t0,0($t1)

 812  beq   $t0,$s5, Exit

 816  addiu $s3,$s3,1

 820  j     Loop

      Exit:
```

Material from past lectures:
What type of C variable is probably stored in $s6?

Write an equivalent C loop using a→$s3, b→$s5, c→$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

In English, what does this loop do?

Code Answers

Addr	Instruction
800	`Loop: sll $t1,$s3,2`
804	`addu  $t1,$t1,$s6`
808	`lw    $t0,0($t1)`
812	`beq   $t0,$s5, Exit`
816	`addiu $s3,$s3,1`
820	`j     Loop`
	`Exit:`

Material from past lectures:

What type of C variable is probably stored in $s6?

int * (or any pointer)

Write an equivalent C loop using a→$s3, b→$s5, c→$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

```
int a,b,*c;
/* values initialized */
while(c[a] != b)  a++;
```

In English, what does this loop do?

Finds an entry in array c that matches b.

Assembly Practice Question

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2
__:
 804   addu  $t1,$t1,$s6
__:
 808   lw    $t0,0($t1)
__:
 812   beq   $t0,$s5, Exit
__:
 816   addiu $s3,$s3,1
__:
 820   j     Loop
__:
       Exit:
```

Assembly Practice Answer (1/4)

`Addr  Instruction`

`800  Loop: sll $t1,$s3,2`

R:

opcode	rs	rt	rd	shamt	funct

`804  addu  $t1,$t1,$s6`

R:

opcode	rs	rt	rd	shamt	funct

`808  lw    $t0,0($t1)`

I:

opcode	rs	rt	immediate

`812  beq   $t0,$s5, Exit`

I:

opcode	rs	rt	immediate

`816  addiu $s3,$s3,1`

I:

opcode	rs	rt	immediate

`820  j     Loop`

J:

opcode	target address

`      Exit:`

Assembly Practice Answer (2/4)

```
Addr  Instruction
 800  Loop: 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  beq   $t0,$s5, Exit
```

I:	4	8	21	2

```
 816  addiu $s3,$s3,1
```

I:	8	19	19	1

```
 820  j     Loop
```

J:	2	200

```
      Exit:
```

Assembly Practice Answer (3/4)

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2
```

R:	000000	00000	10011	01001	00010	000000

```
 804  addu  $t1,$t1,$s6
```

R:	000000	01001	10110	01001	00000	100001

```
 808  lw    $t0,0($t1)
```

I:	100011	01001	01000	0000 0000 0000 0000

```
 812  beq   $t0,$s5, Exit
```

I:	000100	01000	10101	0000 0000 0000 0010

```
 816  addiu $s3,$s3,1
```

I:	001000	10011	10011	0000 0000 0000 0001

```
 820  j     Loop
```

J:	000010	00 0000 0000 0000 0000 1100 1000

```
      Exit:
```

Assembly Practice Answer (4/4)

```
Addr  Instruction
 800  Loop: sll $t1,$s3,2
R:    0x 0013 4880
 804  addu  $t1,$t1,$s6
R:    0x 0136 4821
 808  lw    $t0,0($t1)
I:    0x 8D28 0000
 812  beq   $t0,$s5, Exit
I:    0x 1115 0002
 816  addiu $s3,$s3,1
I:    0x 2273 0001
 820  j     Loop
J:    0x 0800 00C8
      Exit:
```

Agenda

- I-Format
  - Branching and PC-Relative Addressing
- Administrivia
- J-Format
- Pseudo-instructions
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Disassembly Practice

- Disassembly is the opposite process of figuring out the instructions from the machine code
- On the following slides, there are 6-lines of machine code (hex numbers)
- Your task:
  1) Convert to binary
  2) Use opcode to determine format and fields
  3) Write field values in decimal
  4) Convert fields MIPS instructions (try adding labels)
  5) Translate into C (be creative!)
- Use your Green Sheet; answers follow

Disassembly Practice Question

Address	Instruction
`0x00400000`	`0x00001025`
`...`	`0x0005402A`
	`0x11000003`
	`0x00441020`
	`0x20A5FFFF`
	`0x08100001`

Disassembly Practice Answer (1/9)

Address	Instruction
`0x00400000`	`00000000000000000001000000100101`
`...`	`00000000000001010100000000101010`
	`00010001000000000000000000000011`
	`00000000010001000001000000100000`
	`00100000101001011111111111111111`
	`00001000000100000000000000000001`

1) Converted to binary

Disassembly Practice Answer (2/9)

Address		Instruction					
0x00400000	R	000000	00000	00000	00010	00000	100101
...	R	000000	00000	00101	01000	00000	101010
	I	000100	01000	00000	0000000000000011		
	R	000000	00010	00100	00010	00000	100000
	I	001000	00101	00101	1111111111111111		
	J	000010	00000100000000000000000001				

2) Check opcode for format and fields...

– 0 (R-Format), 2 or 3 (J-Format), otherwise (I-Format)

Disassembly Practice Answer (3/9)

Address		Instruction					
0x00400000	R	0	0	0	2	0	37
...	R	0	0	5	8	0	42
	I	4	8	0	+3		
	R	0	2	4	2	0	32
	I	8	5	5	-1		
	J	2	0x0100001				

3) Convert to decimal

– Can leave target address in hex

Disassembly Practice Answer (4/9)

Address	Instruction
`0x00400000`	`or   $2,$0,$0`
`0x00400004`	`slt  $8,$0,$5`
`0x00400008`	`beq  $8,$0,3`
`0x0040000C`	`add  $2,$2,$4`
`0x00400010`	`addi $5,$5,-1`
`0x00400014`	`j    0x0100001`
`0x00400018`	

4) Translate to MIPS instructions (write in addrs)

Disassembly Practice Answer (5/9)

```
Address     Instruction
0x00400000  or   $v0,$0,$0
0x00400004  slt  $t0,$0,$a1
0x00400008  beq  $t0,$0,3
0x0040000C  add  $v0,$v0,$a0
0x00400010  addi $a1,$a1,-1
0x00400014  j    0x0100001 # addr: 0x0400004
0x00400018
```

4) Translate to MIPS instructions (write in addrs)

- More readable with register names

Disassembly Practice Answer (6/9)

```
Address        Instruction
0x00400000           or   $v0,$0,$0
0x00400004     Loop: slt  $t0,$0,$a1
0x00400008           beq  $t0,$0,Exit
0x0040000C           add  $v0,$v0,$a0
0x00400010           addi $a1,$a1,-1
0x00400014           j    Loop
0x00400018     Exit:
```

4) Translate to MIPS instructions (write in addrs)

– Introduce labels

Disassembly Practice Answer (7/9)

Address Instruction

```
        or   $v0,$0,$0    # initialize $v0 to 0
Loop:   slt  $t0,$0,$a1   # $t0 = 0 if 0 >= $a1
        beq  $t0,$0,Exit  # exit if $a1 <= 0
        add  $v0,$v0,$a0  # $v0 += $a0
        addi $a1,$a1,-1   # decrement $a1
        j    Loop
Exit:
```

4) Translate to MIPS instructions (write in addrs)

– What does it do?

Disassembly Practice Answer (8/9)

```
/* a→$v0, b→$a0, c→$a1 */
a = 0;
while(c > 0) {
  a += b;
  c--;
}
```

5) Translate into C code

- Initial direct translation

Disassembly Practice Answer (9/9)

```
/* naïve multiplication: returns m*n */
int multiply(int m, int n) {
  int p; /* product */
  for(p = 0; n-- > 0; p += m) ;
  return p;
}
```

5) Translate into C code

– One of many possible ways to write this