CSE 31

Computer Organization

Lecture 14 – Instruction Format (3)

Announcement

- Project #1
  - Due at 11:59pm on 3/22, Friday (no more late submission)
  - You must demo your submission to your TA during week of 4/1, in lab.
- Lab #7 this week
  - Due after Spring Break
  - Demo your lab within 7 days after due dates
- HW #4 in CatCourses
  - Due Monday (4/1) at 11:59pm
- Reading assignment
  - Chapter 1.6, 6.1-6.3 of zyBooks
    - Make sure to do the Participation Activities
    - Due Wednesday (4/3) at 11:59pm

Review

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use lw and sw).
- 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		

Branch Example

- MIPS Code:

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

decimal representation:

4	9	0	3

binary representation:

000100	01001	00000	0000000000000011

Questions on PC-addressing

- Does the value in branch field change if we move the code?
- What do we do if destination is > 2^{15} instructions away from branch?
- Why do we need different addressing modes (different ways of forming a memory address)? Why not just one?

J-Format Instructions (1/5)

- For branches, we assumed that we won't want to branch too far, so we can specify change in PC.
- For general jumps (`j` and `jal`), we may jump to anywhere in memory.
- Ideally, we could 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, so we compromise.

J-Format Instructions (2/5)

- Define two "fields" of these bit widths:

6 bits	26 bits

- As usual, each field has a name:

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 (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always `00` (in binary).
    - So let's just take this for granted and not even specify them.

J-Format Instructions (4/5)

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to anywhere in memory,
    - but it’s adequate 99.9999…% of the time, since programs aren’t that long
    - only if straddle a 256 MB boundary
  - What if we absolutely need to specify a 32-bit address?
    - We can always put it in a register and use the `jr` instruction.

J-Format Instructions (5/5)

- Summary:
  - New PC = { PC[31..28], target address, 00 }
- Understand where each part came from!
- Note: { , , } means concatenation
  { 4 bits , 26 bits , 2 bits } = 32 bit address
  - { 1010, 11111111111111111111111111, 00 } = 10101111111111111111111111111100

Quiz

When combining two C files into one executable, recall we can compile them independently & then merge them together.

1) Jump insts don’t require any changes.
2) Branch insts don’t require any changes.

	12
a)	FF
b)	FT
c)	TF
d)	TT
e)	dunno

Quiz

When combining two C files into one executable, recall we can compile them independently & then merge them together.

1) Jump insts don't require any changes.
2) Branch insts don't require any changes.

```
   12
a) FF
b) FT
c) TF
d) TT
e)dunno
```

Summary

- MIPS Machine Language Instruction:
  32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use **PC-relative addressing**, Jumps use **absolute addressing**.
- Disassembly is simple and starts by decoding `opcode` field. (more in a week)

Decoding Machine Language

- How do we convert 1s and 0s to assembly language and to C code?

  Machine language ⇒ assembly ⇒ C?

- For each 32 bits:

1. Look at `opcode` to distinguish between R-Format, J-Format, and I-Format.
2. Use instruction format to determine which fields exist.
3. Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.
4. Logically convert this MIPS code into valid C code.

- Always possible? Unique?

Decoding Example (1/7)

- Here are six machine language instructions in hexadecimal:

  $\texttt{00001025}_{\texttt{hex}}$
  $\texttt{0005402A}_{\texttt{hex}}$
  $\texttt{11000003}_{\texttt{hex}}$
  $\texttt{00441020}_{\texttt{hex}}$
  $\texttt{20A5FFFF}_{\texttt{hex}}$
  $\texttt{08100001}_{\texttt{hex}}$

- Let the first instruction be at address $4{,}194{,}304_{ten}$ ($0x00400000_{\texttt{hex}}$).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

```
00000000000000000001000000100101
00000000000001010100000000101010
00010001000000000000000000000011
00000000010001000001000000100000
00100000101001011111111111111111
00001000000100000000000000000001
```

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/7)

- Select the opcode (first 6 bits) to determine the format:

Format:						
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				

- Look at `opcode`:
  0 means R-Format,
  2 or 3 mean J-Format,
  otherwise I-Format.
- Next step: separation of fields

Decoding Example (4/7)

- Fields separated based on format/opcode:

Format:

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	1,048,577				

Next step: translate ("disassemble") to MIPS assembly instructions

Decoding Example (5/7)

- MIPS Assembly (Part 1):

```
Address:              Assembly instructions:
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      0x100001
```

Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)

Decoding Example (6/7)

- MIPS Assembly (Part 2):

```
       or    $v0,$0,$0
Loop:  slt   $t0,$0,$a1
       beq   $t0,$0,Exit
       add   $v0,$v0,$a0
       addi  $a1,$a1,-1
       j     Loop
Exit:
```

Next step: translate to C code (must be creative!)

Decoding Example (7/7)

Before Hex:

00001025$_{hex}$
0005402A$_{hex}$
11000003$_{hex}$
00441020$_{hex}$
20A5FFFF$_{hex}$
08100001$_{hex}$

After C code (Mapping below)

`$v0`: product
`$a0`: multiplicand
`$a1`: multiplier

```
product = 0;
while (multiplier > 0) {
   product += multiplicand;
   multiplier -= 1;
}
```

```
      or   $v0,$0,$0
Loop: slt  $t0,$0,$a1
      beq  $t0,$0,Exit
      add  $v0,$v0,$a0
      addi $a1,$a1,-1
      j    Loop
Exit:
```

Idea: Instructions are just numbers, code is treated like data

Review from before: `lui`

- So how does `lui` (load upper immediate) help us?

  Example:

  ```
  addi    $t0,$t0, 0xABABCDCD
  ```

  becomes:

  ```
  lui     $at, 0xABAB
  ori     $at, $at, 0xCDCD
  add     $t0,$t0,$at
  ```

  - Now each I-format instruction has only a 16-bit immediate.
- Wouldn’t it be nice if the assembler would do this for us automatically?
  - If number too big, then just automatically replace `addi` with `lui`, `ori`, `add`
  - Pseudo-instructions

True Assembly Language (1/3)

- Pseudo-instruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instructions
- What happens with pseudo-instructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
- Some examples follow

Example Pseudo-instructions

- Register Move

```
move     reg2,reg1
```

Expands to:

```
add      reg2,$zero,reg1
```

- Load Immediate

```
li reg,value
```

If value fits in 16 bits:

```
addi     reg,$zero,value
```

else:

```
lui      reg,upper_16_bits_of_value
ori      reg,reg,lower_16_bits
```

Example Pseudo-instructions

- Load Address: How do we get the address of an instruction or global variable into a register?

```
la reg,label
```

Again if value fits in 16 bits:

```
addi     reg,$zero,label_value
```

else:

```
lui      reg,upper_16_bits_of_value
ori      reg,reg,lower_16_bits
```

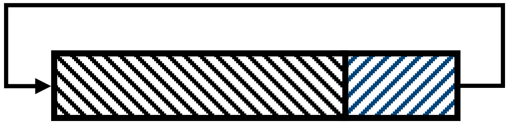
True Assembly Language (2/3)

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses any regular register, it'll overwrite whatever the program has put into it.
- Solution:
  - Reserve a register (`$1`, called `$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.

Example Pseudo-instructions

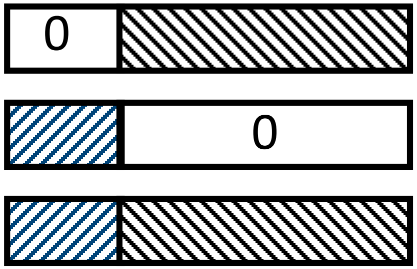
- Rotate Right Instruction

```
 ror      reg, value
```



Expands to:

```
srl      $at, reg, value
sll      reg, reg, 32-value
or       reg, reg, $at
```



- “No OPeration” instruction

```
nop
```

  - Expands to instruction = 0_{ten},
  - `sll  $0, $0, 0`

Example Pseudo-instructions

- Wrong operation for operand

  ```
  addu      reg,reg,value
  ```

  //should be **addiu**

  If value fits in 16 bits, `addu` is changed to:

  ```
  addiu     reg,reg,value
  ```

  else:

  ```
  lui $at,upper 16 bits of value
  ori $at,$at,lower 16 bits
  addu      reg,reg,$at
  ```

- How do we avoid confusion about whether we are talking about MIPS assembler with or without pseudo-instructions?

True Assembly Language (3/3)

- MAL (MIPS Assembly Language):
  - The set of instructions that a programmer may use to code in MIPS; this includes pseudo-instructions
- TAL (True Assembly Language):
  - Set of instructions that can actually get translated into a single machine language instruction (32-bit binary string)
- A program must be converted from MAL into TAL before translation into 1s & 0s.

Questions on Pseudo-instructions

- Question:
  - How does MIPS assembler / SPIM (MIPS simulator) recognize pseudo-instructions?
- Answer:
  - It looks for officially defined pseudo-instructions, such as `ror` and `move`
  - It looks for special cases where the operand is incorrect for the operation and tries to handle it gracefully

Rewrite TAL as MAL

- TAL:

```
        or    $v0,$0,$0
Loop:   slt   $t0,$0,$a1
        beq   $t0,$0,Exit
        add   $v0,$v0,$a0
        addi  $a1,$a1,-1
        j     Loop
Exit:
```

- This time convert to MAL
- It’s OK for this exercise to make up MAL instructions

Rewrite TAL as MAL (Answer)

TAL:

```
        or   $v0,$0,$0
Loop:   slt  $t0,$0,$a1
        beq  $t0,$0,Exit
        add  $v0,$v0,$a0
        addi $a1,$a1,-1
        j    Loop
 Exit:
```

MAL:

```
              li      $v0,0
    Loop:     ble     $a1,$zero,Exit
              add     $v0,$v0,$a0
              sub     $a1,$a1,1
              j       Loop
    Exit:
```

Quiz

- Which of the instructions below are MAL and which are TAL?

1. addi $t0, $t1, 40000
2. beq $s0, 10, Exit

	12
a)	MM
b)	MT
c)	TM
d)	TT

Quiz

- Which of the instructions below are MAL and which are TAL?

1. addi $t0, $t1, 40000
2. beq $s0, 10, Exit

	12
a)	**MM**
b)	MT
c)	TM
d)	TT

Summary

- Disassembly is simple and starts by decoding `opcode` field.
  - Be creative and efficient when authoring C
- Assembler expands real instruction set (TAL) with pseudo-instructions (MAL)
  - Only TAL can be converted to raw binary
  - Assembler's job to do conversion
  - Assembler uses reserved register `$at`
  - MAL makes it <u>much</u> easier to write MIPS