CSE 31
Computer Organization

Lecture 21 – Instruction Format (wrap up)

Announcements

- Labs
  - Lab 7 grace period* ends this week
    - » Demo is REQUIRED to receive full credit
  - Lab 8 due this week (with **7 days grace period*** after due date)
    - » Demo is REQUIRED to receive full credit
  - Lab 9 out this week
    - » Due at 11:59pm on the same day of your lab after next (with 7 days grace period* after due date)
    - » You must demo your submission to your TA within 14 days from posting of lab
    - » Demo is REQUIRED to receive full credit
- Reading assignments
  - Reading 07 (zyBooks 5.1 – 5.6) due 17-APR
    - » Complete **Participation Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

* A 10% penalty will be applied for late *submissions* during the grace period.

Announcements

- Homework assignment
  - Homework 06 (zyBooks 5.1 – 5.11) due 01-MAY
    - » Complete **Challenge Activities** in each section to receive grade
    - » IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- Midterm 02
  - On 19-APR during class
  - Announcement and Sample Exam posted on CatCourses
- Project 02
  - Due 05-MAY
  - Can work in teams of 2 students
    - » Each team member must identify teammate in "Comments…" text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - **No grace period**
    - » **Must complete submission and demo by due date.**

I-Format Problem (review)

- Problem:
  - Need a way to deal with a 32-bit immediate in any I-format instruction.
- Solution
  - New instruction: `lui register, immediate`
  - Takes 16-bit immediate and puts these bits in the upper half (high order half) of the register and sets lower half to `0`s
- Example:

```
        addiu $t0,$t0, 0xABABCDCD
```

… becomes

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

$\boxed{\texttt{addu \$t0,\$t0,\$at}}$ R-format!

Branches: PC-Relative Addressing (1/5)

- Use I-Format for branch instructions as well

opcode	rs	rt	immediate

- `opcode` specifies `beq` versus `bne`
- `rs` and `rt` specify registers to compare
- What can immediate specify?
  - `immediate` is only 16 bits
  - PC (Program Counter) has byte address of current instruction being executed;
    - » 32-bit pointer to memory
  - So `immediate` cannot specify entire address to branch to.

Branches: PC-Relative Addressing (2/5)

- How do we typically use branches?
  - Answer: `if-else`, `while`, `for`
  - Loops are generally small: usually up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (`j` and `jal`), not the branches.

- Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount

Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction:
  - PC-Relative Addressing
- Let the 16-bit immediate field be a signed two's complement integer to be added to the PC if we take the branch.
- Now we can branch $\pm 2^{15}$ bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with `00` in binary).
  - So, the number of bytes to add to the PC will always be a multiple of 4.
  - So, specify the `immediate` in words.

- Now, we can branch $\pm 2^{15}$ words from the PC (or $\pm 2^{17}$ bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don’t take the branch:
    PC = PC + 4 = byte address of next instruction
  - If we do take the branch:
    PC = (PC + 4) + (`immediate` * 4) ← Why multiply by 4?
  - Observations
    - » `Immediate` field specifies the number of words to jump, which is simply the number of instructions to jump.
    - » `Immediate` field can be positive or negative.
    - » Due to hardware, add `immediate` to (PC+4), not to PC; will be clearer (why? later in course)

Branch Example (1/3)

- MIPS Code:

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

- `beq` branch is I-Format:

  `opcode` = 4 (look up in table – MIPS Reference Sheet)

  `rs` = 9 (first operand)

  `rt` = 0 (second operand)

  `immediate` = ??? (Do we need to know the address of End?)

Branch Example (2/3)

- MIPS Code:

```
1000 Loop: beq   $9,$0,End   ← PC
1004       addu  $8,$8,$10   ← PC + 4
1008       addiu $9,$9,-1
100c       j     Loop
1010 End:
```

- `immediate` **Field:**
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.
  - In `beq` case, `immediate` = 3
  - PC = (PC + 4) + (`immediate` * 4)

    = (1000 + 4) + (3 * 4)

    = 1004 + C

    = 1010

Branch Example (3/3)

- 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 (we can compile them independently and then merge them together):

1) Jump instructions don't require any changes.
2) Branch instructions 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 (we can compile them independently and then merge them together):

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

Decoding Machine Language

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

  Machine language $\Rightarrow$ assembly $\Rightarrow$ 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 $4{,}194{,}304_{\text{ten}}$ (i.e., $\text{0x0040 0000}_{\texttt{hex}}$).
- 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 (values shown in Hex, except address and immediate fields):

R	0	0	0	2	0	25
R	0	0	5	8	0	2A
I	4	8	0	+3		
R	0	2	4	2	0	20
I	8	5	5	-1		
J	2	00000100000000000000000001				

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     0x00400004`

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:

```
addu      reg2, $zero, reg1
```

- Load Immediate

```
li        reg, value
```

If value fits in 16 bits:

```
addiu     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
```

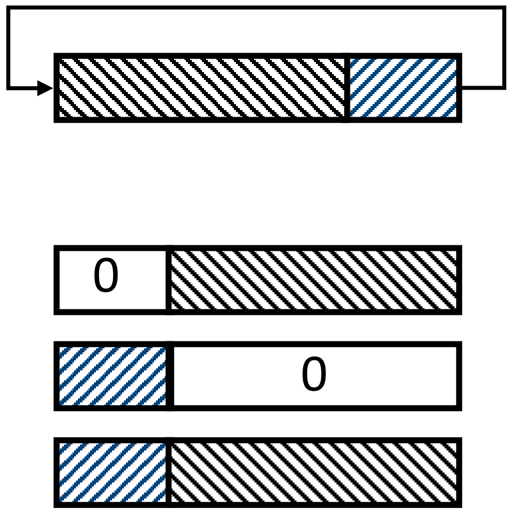
True Assembly Language (2/3)

- 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, 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