inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 17 – Introduction to MIPS
Instruction Representation III

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Digital film network ⇒
The UK is investing in 150 digital cinemas! Each will get a 100 GiB lossless digital copy of the film and show it on digital 2K (2048x1080) projectors. USA?!

news.bbc.co.uk/1/hi/technology/4297865.stm

Clarification - IEEE Four Rounding Modes

- **<u>This is just an example</u> in base 10 to show you the 4 modes.**
- **What really happens is…**

1) in **binary**, not decimal!
2) at the lowest bit of the mantissa with the **guard bit(s)** as our extra bit(s), and you need to decide how these extra bit(s) affect the result if the guard bits are "**100…**"
3) If so, you're half-way between the representable numbers.

E.g., 0.10**10** is 5/8, halfway between our representable 4/8 [1/2] and 6/8 [3/4]. Which number do we round to? 4 modes!

Outline

- **Disassembly**

- **Pseudoinstructions and "True" Assembly Language (TAL) v. "MIPS" Assembly Language (MAL)**

Decoding Machine Language

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

  Machine language ⟹ C?

- **For each 32 bits:**

  - **Look at `opcode`: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format.**

  - **Use instruction type to determine which fields exist.**

  - **Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number.**

  - **Logically convert this MIPS code into valid C code. Always possible? Unique?**

Decoding Example (1/7)

- **Here are six machine language instructions in hexadecimal:**

  00001025_{hex}
  $0005402A_{hex}$
  11000003_{hex}
  00441020_{hex}
  $20A5FFFF_{hex}$
  08100001_{hex}

- **Let the first instruction be at address $4,194,304_{ten}$ ($0x00400000_{hex}$).**

- **Next step: convert hex to binary**

Decoding Example (2/7)

- **The six machine language instructions in binary:**

```
00000000000000000001000000100101
00000000000010101000000000101010
00010001000000000000000000000011
00000000100010000010000000100000
00100001010010111111111111111111
00001000000100000000000000000001
```

- **Next step: identify opcode and format**

R	0	rs	rt	rd	shamt	funct
I	1, 4-31	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 00000000000001000000100101
R 000000 00000010101000000000101010
I 000100 01000000000000000000000011
R 000000 00100010000010000000100000
I 001000 00101001011111111111111111
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:**

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)

- **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 (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:
```

Demonstrated Big 61C Idea: Instructions are just numbers, code is treated like data

Administrivia

- **Thanks to TAs who filled in last week**

- **SIGCSE 2005 was GREAT**

- **Your midterm is in 7 days!**

Review from before: `lui`

- **So how does `lui` 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 this for us automatically?**
  - **If number too big, then just automatically replace `addi` with `lui, ori, add`**

True Assembly Language (1/3)

- **Pseudoinstruction: A MIPS instruction that doesn't turn directly into a machine language instruction, but into other MIPS instrucitons**

- **What happens with pseudoinstructions?**
  - **They're broken up by the assembler into several "real" MIPS instructions.**
  - **But what is a "real" MIPS instruction? Answer in a few slides**

- **First some examples**

Example Pseudoinstructions

- ## Register Move
  `move    reg2,reg1`
  Expands to:
  `add     reg2,$zero,reg1`

- ## Load Immediate
  `li      reg,value`
  If value fits in 16 bits:
  `addi    reg,$zero,value`
  else:
  `lui     reg,upper 16 bits of value`
  `ori     reg,$zero,lower 16 bits`

True Assembly Language (2/3)

- **Problem:**
  - **When breaking up a pseudoinstruction, the assembler may need to use an extra reg.**
  - **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 Pseudoinstructions

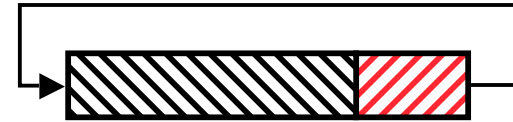- ## Rotate Right Instruction

  `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 Pseudoinstructions

- ## 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 pseudoinstructions?

True Assembly Language (3/3)

- **MAL** (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this <u>includes</u> pseudoinstructions

- **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 Pseudoinstructions

- **Question:**
  - **How does MIPS 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:   bge   $zero,$a1,Exit
        add   $v0,$v0,$a0
        sub   $a1,$a1,1
        j     Loop
Exit:
```

Peer Instruction

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

A. `addi $t0, $t1, 40000`

B. `beq  $s0, 10, Exit`

C. `sub  $t0, $t1, 1`

```
     ABC
1:   MMM
2:   MMT
3:   MTM
4:   MTT
5:   TMM
6:   TMT
7:   TTM
8:   TTT
```

In conclusion

- **Disassembly is simple and starts by decoding `opcode` field.**
  - **Be creative, efficient when authoring C**

- **Assembler expands real instruction set (TAL) with pseudoinstructions (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**