CS61C – Machine Structures

Lecture 17 - MIPS Instruction Representation III

2/27/2006

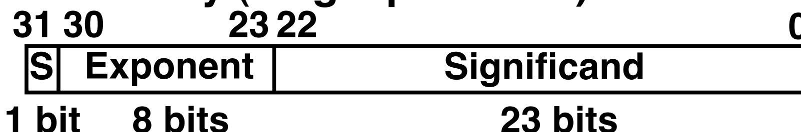
John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

IEEE 754 Floating Point Standard (review)

- **Biased Notation**, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single precision
  - Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision
- **Summary (single precision):**



$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

Double precision exp:11, significand:52 and exponent bias of 1023

IEEE 754 Floating Point Review (2)

- Encodings (Single Precision):

Exponent	Significand	Object
0	0	0
0	nonzero	denormal.
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Denormalized number: no (implied) leading 1, exponent = -126.

Wawrzynek Spring 2006 © UCB

Outline

- Disassembly
- Pseudoinstructions and
“True” Assembly Language (TAL) v.
“MIPS” Assembly Language (MAL)

Wawrzynek Spring 2006 © UCB

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 distinquish 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?

Wawrzynek Spring 2006 © UCB

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 $4{,}194{,}304_{\text{ten}}$ ($\text{0x00400000}_{\text{hex}}$).
- Next step: convert hex to binary

Wawrzynek Spring 2006 © UCB

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, 4-31	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  00000000000000000001000000100101
R  00000000000001010100000000101010
I  00010001000000000000000000000011
R  00000000010001000001000000100000
I  00100000101001011111111111111111
J  00001000000100000000000000000001
```

- 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:

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**

Wawrzynek Spring 2006 © UCB

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)**

Wawrzynek Spring 2006 © UCB

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!)

Wawrzynek Spring 2006 © UCB

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

Wawrzynek Spring 2006 © UCB

Administrivia

- Exam ready to return
- Stats posted on website
- Regrade policy on website:
  - Put it in writing
  - Before one week from today

Wawrzynek Spring 2006 © UCB

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`

Wawrzynek Spring 2006 © UCB

True Assembly Language (1/3)

- **Pseudoinstruction**: 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

Wawrzynek Spring 2006 © UCB

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
```


Wawrzynek Spring 2006 © UCB

Example Pseudoinstructions

° 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,$zero,lower 16 bits
```


Wawrzynek Spring 2006 © UCB

True Assembly Language (2/3)

° **Problem:**

- When breaking up a pseudo-instruction, 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.

Wawrzynek Spring 2006 © UCB

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
```


Wawrzynek Spring 2006 © UCB

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?

Wawrzynek Spring 2006 © UCB

True Assembly Language (3/3)

- **MAL (MIPS Assembly Language): the set of instructions that a programmer may use to code in MIPS; this includes 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 assembler / SPIM 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:
```

Quiz

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

Wawrzynek Spring 2006 © UCB

Quiz Answer

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

i. addi $t0, $t1, 40000 — 40,000 > +32,767 =>`lui`,`ori`

ii. beq $s0, 10, Exit — `Beq:` both must be registers; Exit: if > 2^{15}, then MAL

iii. sub $t0, $t1, 1 — `sub:` both must be registers; even if it was subi, there is no subi in TAL; generates addi $t0,$t1, -1

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

Wawrzynek Spring 2006 © UCB

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 much easier to write MIPS**

Wawrzynek Spring 2006 © UCB