inst.eecs.berkeley.edu/~cs61c

# UC Berkeley CS61C : Machine Structures

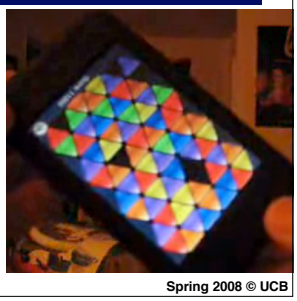
## Lecture 17
## Instruction Representation III

**2008-03-03**

**TA Matt Johnson**

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

**iPhone games! (and general SDK) ⇒**



**Apple is (finally) releasing an iPhone Software Developer Kit on March 6th (?) That means iPhone games that use both touch and accelerometer input!**

youtube.com/watch?v=hy0ptZisr70



## Review

- **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.**

## Outline

- Disassembly
- Pseudoinstructions
- "True" Assembly Language (TAL) vs. "MIPS" Assembly Language (MAL)



## 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:

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

- 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
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:
```

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



## Administrivia

- **Midterm is next week! Day and location are still TBA**
  - **Old midterms online (link at top of page)**
  - **Lectures and reading materials fair game**
  - **Fix green sheet errors (if old book)**
- **Review session also TBA**
- **Project 2 is due March 5 at 11:59PM**
  - **That's Wednesday!**
  - **There was a file update. See spec page.**

## 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 instructions
- What happens with pseudo-instructions?
  - They're broken up by the assembler into several "real" MIPS instructions.
- Some examples follow

## 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
```



## 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
```

## 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.



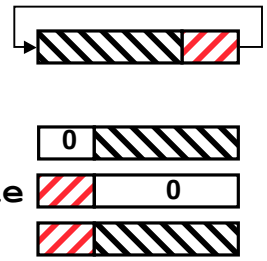
## Example Pseudoinstructions

- 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 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:**

```
      Loop:   or    $v0,$0,$0
              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:
```



## Peer Instruction

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

i. addi $t0, $t1, 40000

ii. beq $s0, 10, Exit

iii. sub $t0, $t1, 1

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

## Peer Instruction 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 |



## 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