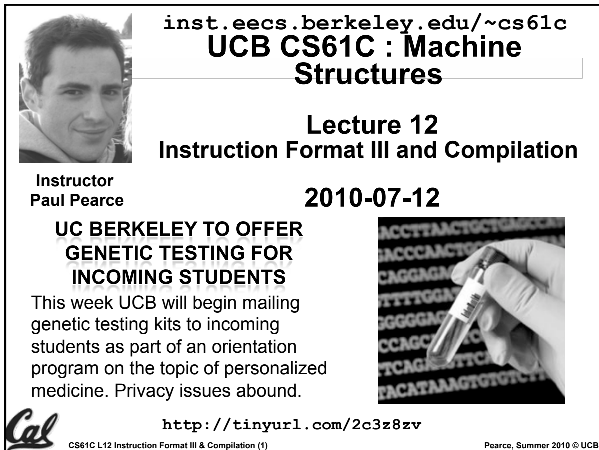


In Review

- 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.
- Disassembly is simple and starts by decoding `opcode` field. (Right now!)

Outline

- Disassembly
- Pseudo-instructions
- “True” Assembly Language (TAL) vs. “MIPS” Assembly Language (MAL)
- Begin discussing Compilation

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:

  $\texttt{00001025}_{hex}$
  $\texttt{0005402A}_{hex}$
  $\texttt{11000003}_{hex}$
  $\texttt{00441020}_{hex}$
  $\texttt{20A5FFFF}_{hex}$
  $\texttt{08100001}_{hex}$
- Let the first instruction be at address $4{,}194{,}304_{ten}$ ($\texttt{0x00400000}_{hex}$).
- Next step: convert hex to binary

Decoding Example (2/7)

- The six machine language instructions in binary:

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

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
0x00400018	

- **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    #t0 = 1 if $0 < $a0
                            #t0 = 0 if $0>= $a0
        beq   $t0,$0,Exit   # goto exit
                            # if $a0 <= 0
        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

`$v0`: product `$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

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`

Administrivia

- Midterm is Friday! 9:30am-12:30 in 100 Lewis!
  - Midterm covers material up to and including Tuesday July 13th.
  - Old midterms online (link at top of page)
  - Lectures and reading materials fair game
  - Bring 1 sheet of notes (front and back) and a pencil. We'll provide the green sheet.
- Review session tonight, 6:30pm in 306 Soda
- There are "CS Illustrated" posters on floating point at the end of today's handout.
  - Be sure to check them out!

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 1 or more "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
```

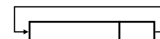
True Assembly Language (2/3)

- 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

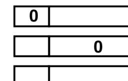
- Rotate Right Instruction

```
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 (which exist in the MIPS ISA) that can actually get directly translated into a single machine language instruction (32-bit binary string). Green sheet is TAL!
- A program must be converted from MAL into TAL before translation into 1s & 0s.

Questions on Pseudoinstructions

- Question:
  - How does MIPS assembler / Mars 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 # goto exit
                          # if $a0 <= 0
        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:
```

Review

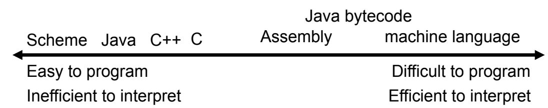
- Disassembly is simple and starts by decoding **opcode** field.
  - Be creative, efficient when authoring C
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  - Only TAL can be converted to raw binary
  - Assembler's job to do conversion
  - Assembler uses reserved register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

Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler (next time)
  - Assembler (next time)
  - Linker (next time)
  - Loader (next time)
- An Example (next time)

Language Execution Continuum

- An Interpreter is a program that executes other programs.

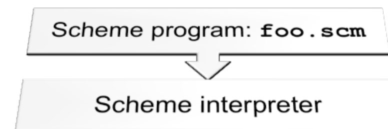


- Language translation gives us another option.
- In general, we interpret a high level language when efficiency is not critical and translate to a lower level language to up performance

Interpretation vs Translation

- How do we run a program written in a source language?
  - Interpreter: Directly executes a program in the source language
  - Translator: Converts a program from the source language to an equivalent program in another language
- For example, consider a Scheme program **`foo.scm`**

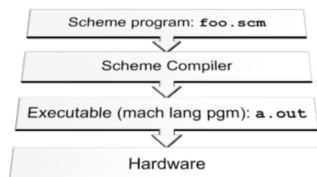
Interpretation



- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

Translation

- Scheme Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpeter of machine language.



Interpretation

- Any good reason to interpret machine language in software?
- MARS– useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to x86.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

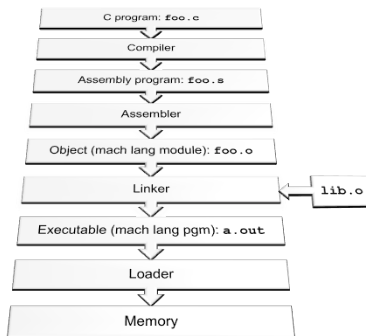
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS, stk)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (eg. Microsoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

Steps to Starting a Program (translation)



Peer Instruction

- 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
```

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
- Interpretation vs translation

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

jump example (1/5)

address (shown in hex)

```
   2345ABC4 addi $s3,$zero,1016
PC 2345ABC8 j    LABEL
   2345ABCC add  $t0, $t0, $t0
   …
   2ABCDE10 LABEL: add $s0,$s0,$s1
   …
```

- `j` J-Format:
  - `opcode` = 2 (look up in table)
  - `target address = ???`

jump example (2/5)

address (shown in hex)

```
   2345ABC4 addi $s3,$zero,1016
PC 2345ABC8 j    LABEL
   2345ABCC add  $t0, $t0, $t0
   …
   2ABCDE10 LABEL: add $s0,$s0,$s1
   …
```

Note: The first 4 bits of PC+4 and the target are the same! That's why we can do this!

- `j` J-Format:

We want to jump to 0x2ABCDE10. As binary:

Target address

31 … 0

0010	10101011110011011110000100	00

jump example (3/5)

address (shown in hex)

```
   2345ABC4 addi $s3,$zero,1016
PC 2345ABC8 j    LABEL
   2345ABCC add  $t0, $t0, $t0
   …
   2ABCDE10 LABEL: add $s0,$s0,$s1
   …
```

- `j` J-Format:

binary representation:

000010	10101011110011011110000100

hexadecimal representation: `0AAF 3784`$_{hex}$

jump example (4/5)

- J How do we reconstruct the PC?:

address (shown in hex)

```
   2345ABC4 22D5 FFCEhex # addi …
PC 2345ABC8 0AAF 3784hex # jump …
   2345ABCC 012A 4020hex # add …
   …
```

Machine level Instruction (binary representation):

000010	10101011110011011110000100

Jump — Target address

jump example (5/5)

- J How do we reconstruct the PC?:

address (shown in hex)

```
   2345ABC4 22D5 FFCEhex # addi …
PC 2345ABC8 0AAF 3784hex # jump …
   2345ABCC 012A 4020hex # add …
   …
```

New PC = { (PC+4)[31..28], target address, 00 }

Target address

31 … 0

0010	10101011110011011110000100	00