Today (10/6/2008)

- HW #1 – due today at 5pm, submit via catalyst
- Lab #1 posted

- **Machine language**, the binary representation for instructions.
  - We'll see how it is **designed for the common case**
    - Fixed-sized (32-bit) instructions
    - Only 3 instruction formats
    - Limited-sized immediate fields

Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.
- MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

- Register-to-register arithmetic instructions use the R-type format.

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- This format includes six different fields.
  - op is an operation code or opcode that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is only used for shift instructions.
  - func is used together with op to select an arithmetic instruction.
- The inside back cover of the textbook lists opcodes and function codes for all of the MIPS instructions.

About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111.
  - For example, $t8 is register $24, which is represented as 11000.
  - The complete mapping is given on page A-23 in the book.
- The number of registers available affects the instruction length.
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word.
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations.

I-type format

- Load, store, branch and immediate instructions all use the I-type format.

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- For uniformity, op, rs and rt are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rt is a source register for branches, but a destination register for the other I-type instructions.
- The address is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767.
  - But that's not always enough!

Larger constants

- Larger constants can be loaded into a register 16 bits at a time.
  - The load upper immediate instruction lui loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0000 0000 0011 1101 0000 1001 0000 0000:

```
lui $s0, 0x003D        # $s0 = 003D 0000 (in hex)
ori $s0, $s0, 0x0900   # $s0 = 003D 0900
```

- This illustrates the principle of making the common case fast.
  - Most of the time, 16-bit constants are enough.
  - It's still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.
- Pseudo-instructions may contain large constants. Assemblers including SPIM will translate such instructions correctly.

Loads and stores

- The limited 16-bit constant can present problems for accesses to global data.
- Suppose we want to load from address 0x10010004.

```
lui  $at, 0x1001          # 0x1001 0000
lw   $t1, 0x0004($at)     # Read from Mem[0x1001 0004]
```


Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

```
        beq  $at, $0, L
        add  $v1, $v0, $0
        add  $v1, $v1, $v1
        j    Somewhere
  L:    add  $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:

000100	00001	00000	0000 0000 0000 0011
op	rs	rt	address

- For some reason SPIM is off by one, so the code it produces would contain an address of 4. (But SPIM branches still execute correctly.)

Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
        beq  $s0, $s1, Far
        ...
```

can be simulated with the following actual code.

```
        bne  $s0, $s1, Next
        j    Far
Next:   ...
```

- Again, the MIPS designers have taken care of the common case first.

J-type format

- Finally, the jump instruction uses the J-type instruction format.

op	address
6 bits	26 bits

- The jump instruction contains a *word* address, not an offset
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4000," it's enough to just say "jump to instruction 1000."
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.
    - your MP solutions had better be smaller than 256MB
- For even longer jumps, the jump register, or jr, instruction can be used.

```
jr  $ra      # Jump to 32-bit address in register $ra
```

Summary of Machine Language

- Machine language is the binary representation of instructions:
  - The format in which the machine actually executes them
- MIPS machine language is designed to simplify processor implementation
  - Fixed length instructions
  - 3 instruction encodings: R-type, I-type, and J-type
  - Common operations fit in 1 instruction
    - Uncommon (e.g., long immediates) require more than one

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

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, JFormat, 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 (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 $4{,}194{,}304_{ten}$ (0x00400000hex)
- Next step: convert hex to binary

Decoding (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 (3/7)

- Select the opcode (first 6 bits) to determine the format:

  00000000000000000001000000100101
  00000000000001010100000000101010
  00010001000000000000000000000011
  00000000010001000001000000100000
  00100000101001011111111111111111
  00001000000100000000000000000001

- Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
- Next step: separation of fields R R I R I J Format:

R	0	rs	rt	rd	shamt	funct
I	1, 4-62	rs	rt	immediate		
J	2 or 3	target address				

15

Decoding (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") MIPS assembly instructions R R I R I J Format:

16

Decoding (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)

17

Decoding (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!)

18

Decoding (7/7)

- Possible C code:

```
$v0: var1
$a0: var2
$a1: var3
var1 = 0;
while (var3 > 0) {
  var1 += var2;
  var3 -= 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:
```