Lecture 6

- Happy April Fools
- Finish up functions/stack
- Machine language, the binary representation for instructions.
  - We'll see how it is designed for the common case
  - Fixed-sized (16-bit) instructions
  - Only 3 instruction formats
  - Limited-sized immediate fields

Review

- Why is jal special?
- Why do we need to save registers?
- Why a stack?
- What is the stack used for?

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - The stack grows downward in terms of memory addresses.
  - The address of the top element of the stack is stored by convention in the "stack pointer" register, $sp$.
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.

Accessing and popping elements

- You can access any element in the stack (not just the top one) if you know where it is relative to $sp$.
- For example, to retrieve the value of $s1$:
  - $t0 = s0, 4(s6p)$
- You can pop, or "erase," elements simply by adjusting the stack pointer upwards.
- To pop the value of $s1$, yielding the stack shown at the bottom:
  - $addi \ s6p, \ s6p, \ 4$
- Note that the popped data is still present in memory, but data past the stack pointer is considered invalid.

Pushing elements

- To push elements onto the stack:
  - Move the stack pointer $s6p$ down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $s1$ and $s2$ onto the stack:
  - $addi \ s6p, \ s6p, \ 4$
  - $s1, \ s1, \ \times(s6p)$
  - $s2, \ s2, \ 8(s6p)$
- An equivalent sequence is:
  - $s1, \ s1, \ -(s6p)$
  - $s2, \ 8(s6p)$
- Before and after diagrams of the stack are shown on the right.

Example
Stack Summary

- We just focused on implementing function calls in HIPS.
  - We call functions using JAL, passing arguments in registers $s0-$s4.
  - Functions place results in $s0-$s4 and return using jr $ra$.
- Managing resources is an important part of function calls.
- To keep important data from being overwritten, registers are saved according to conventions for callee-save and callee-call registers.
- Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- Assembly programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.

Assembly vs. machine language

- So far we’ve been using assembly language.
  - We assign names to operations (e.g., ADD) and operands (e.g., 50h).
- Branches and jumps use labels instead of actual addresses.
- Assembly programs 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.
- HIPS machine language is designed to be easy to decode.
- Each HIPS instruction is the same length, 32 bits.
- There are only three different instruction formats, which are very similar to each other.
- Studying HIPS 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.

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td></td>
</tr>
</tbody>
</table>

- 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 green card in the textbook lists opcodes and function codes for all of the HIPS instructions.

L-type format

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

<table>
<thead>
<tr>
<th>op</th>
<th>rt</th>
<th>rd</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- For uniformity, op, rt, and rd are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rt is a source register or an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rd is a source register for branches and stores, but a destination register for the other l-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!

Two’s complement (reminder)

- Easy to do in HW
  - Most significant bit tells sign (sign bit)
- Addition can be done without anything special
- How?
  - Invert all bits and add one
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 0.
  - An immediate logical OR, ori, then sets the lower 16 bits.
- To load the 32-bit value 0x0000 0001 1101 0000 1011 0000 0000:

  
  ```
  lui $t0, 0x0000 0001 1101 0000
  ori $t0, $t0, 0x1001 0000
  ```

- 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.

---

Branches

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

  ```
  beq Sat, 0x2006, l
  add $v1, $v0, 0
  add $v1, $v1, $v0
  beq Sat, 0x2007, l
  ```

- 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:

  
  ```
  000000 0011 0100 0000 0000 0000 0000 0007
  ```

- 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.)

---

J-type format

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

  ```
  000000 0000 0000 0000 0000 0000 0000 0000
  ```

- 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 2^26.
  - Your SPIM solutions had better be smaller than 165MB.
- For even longer jumps, the jump register, or jr, instruction can be used.

  ```
  jr $r32 # Jump to 32-bit address in register $r32
  ```

---

Loads and stores

- The limited 16-bit constant can present problems for accesses to global data.
- Suppose we want to load from address 0x10010004, which won’t fit in the 16-bit address field. Solution:

  ```
  lui $t0, 0x1001
  lar $t1, 0x0004($t0) # Read from Mem[0x1001 0004]
  ```

---

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
  ```

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

---

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
Decoding Machine Language

How do we convert 1s and 0s to assembly language and to C code?

Machine language \(\rightarrow\) assembly \(\rightarrow\) 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 (1/7)

- Here are six machine language instructions in hexadecimal:
  - 00001025
  - 0005402A
  - 11000003
  - 00441020
  - 20A5FFFF
  - 08100001

- Let the first instruction be at address 4,194,304
  (0x00400000hex)
- Next step: convert hex to binary

Decoding (2/7)

- The six machine language instructions in binary:
  - 0000000000000000000010001000101
  - 000000000000000000001010101010
  - 0010010000000000000000000000011
  - 000000000000000000000000000000000000
  - 0000100000000000000000000000000000000000001
- Next step: Identify opcode and format

Decoding (3/7)

- Select the opcode (first 6 bits) to determine the format:
  - 000000 00000 00000 00010 00000 00000
  - 000000 00000 00001 01000 00000 00000
  - 000000 00000 00010 00000 00000 00000
  - 000000 00000 00001 00000 00000 00000
  - 000000 00000 00010 00000 00000 00000
  - 000000 00000 00001 00000 00000 00000
  - 000010 00000 00000 00000 00000 00000
- Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
- Next step: separation of fields R I R I J Format:

Decoding (4/7)

- Fields separated based on format/opcode:

Decoding (5/7)

- MIPS Assembly (Part 1):
- Address: Assembly instructions:
  - 0x00400000 or $2,50,50
  - 0x00400004 slt $8,50,55
  - 0x00400008 beq $8,50,3
  - 0x0040000c add $2,52,54
  - 0x00400010 addi $5,55,-1
  - 0x00400014 j 0x10000
- Better solution: translate to more meaningful MIPS instructions (fix the branch/jump and add labels, registers)
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)

Decoding (7/7)

- Possible C code:

$\text{var1}$: 
$\text{var2}$: 
$\text{var3}$:

\[
\begin{align*}
\text{var1} &= 0; \\
\text{while} \ (\text{var3} < 0) \{
\text{var1} &= \text{var2}; \\
\text{var3} &= 1;
\}
\]

Exit:

\text{strlen Example}

```
\text{**Prototype**}:

\text{char*\ strlen(char*)}

\text{\textbf{Example}}:

\text{**Source**}:

\text{\textbf{Program}}:

```

\text{strlen Example}

```
```
\text{**Prototype**}:

\text{char*\ strlen(char*)}

\text{\textbf{Example}}:

\text{**Source**}:

\text{\textbf{Program}}:

```

```