Processor

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 2, 4.1-4.4, Appendices A and B
Administration

Partner finding assignment on CMS

Office hours over break
Goal for Today

Understanding the basics of a processor
We now have enough building blocks to build machines that can perform non-trivial computational tasks

Putting it all together:
• Arithmetic Logic Unit (ALU)—Lab0 & 1, Lecture 2 & 3
• Register File—Lecture 4 and 5
• Memory—Lecture 5
  – SRAM: cache
  – DRAM: main memory
• Instruction types
• Instruction datapaths
MIPS Register File

A Single cycle processor
MIPS register file

- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via \( R_w \)
  - Writes occur on falling edge but only if WE is high
- Read ports indexed via \( R_A \), \( R_B \)

**Register File**

\[ \text{D}_w \text{ Dual-Read-Port} \]
\[ \text{Single-Write-Port} \]
\[ 32 \times 32 \]
MIPS Register file

MIPS register file
- 32 registers, 32-bits each (with r0 wired to zero)
- Write port indexed via \( R_W \)
  - Writes occur on falling edge but only if WE is high
- Read ports indexed via \( R_A \), \( R_B \)

Why 32? Smaller is faster
MIPS Register file

Registers

- Numbered from 0 to 31
- Each register can be referred by number or name
- $0, $1, $2, $3 ... $31
- Or, by convention, each register has a name
  - $16 - $23 → $s0 - $s7
  - $8 - $15 → $t0 - $t7
  - $0 is always $zero
  - P&H
MIPS Memory

A Single cycle processor
MIPS Memory

- 32-bit address
- 32-bit data
  (but byte addressed)
- Enable + 2 bit memory control (mc)
  00: read word (4 byte aligned)
  01: write byte
  10: write halfword (2 byte aligned)
  11: write word (4 byte aligned)
A processor executes instructions
Processor has some internal state in storage elements (registers)
A memory holds instructions and data
Harvard architecture: separate insts and data
von Neumann architecture: combined inst and data
A bus connects the two
A **stored-program computer** is one which stores program instructions in electronic memory. Often the definition is extended with the requirement that the treatment of programs and data in

**To make a computer**

Need a program

- Stored program computer

Architectures

- von Neumann architecture
- Harvard (modified) architecture
The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data. --- http://en.wikipedia.org/wiki/Harvard_architecture

Under pure von Neumann architecture the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. In a computer using the Harvard architecture, the CPU can both read an instruction and perform a data memory access at the same time, even without a cache. A Harvard architecture computer can thus be faster for a given circuit complexity because instruction fetches and data access do not contend for a single memory pathway.

Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four bit value, while data address zero might indicate an eight bit byte that isn’t part of that twenty-four bit value.

A modified Harvard architecture machine is very much like a Harvard architecture machine, but it relaxes the strict separation between instruction and data while still letting the CPU concurrently access two (or more) memory buses. The most common modification includes separate instruction and data caches backed by a common
Takeaway

A processor executes instructions
  • Processor has some internal state in storage elements (registers)
A memory holds instructions and data
  • (modified) Harvard architecture: separate insts and data
  • von Neumann architecture: combined inst and data
A bus connects the two

We now have enough building blocks to build machines that can perform non-trivial computational tasks
Next Goal

How to program and execute instructions on a MIPS processor?
Levels of Interpretation: Instructions

Programs written in a High Level Language
- C, Java, Python, Ruby, ...
- Loops, control flow, variables

Need translation to a lower-level computer understandable format
- **Assembly** is human readable machine language
- Processors operate on **Machine Language**

Machine Implementation

```
for (i = 0; i < 10; i++)
    printf("go cucs");
```
Levels of Interpretation: Instructions

for (i = 0; i < 10; i++)
    printf("go cucs");

main: addi r2, r0, 10
    addi r1, r0, 0
    loop: slt r3, r1, r2
    ...

High Level Language
- C, Java, Python, Ruby, ...
- Loops, control flow, variables

Assembly Language
- No symbols (except labels)
- One operation per statement

Machine Language
- Binary-encoded assembly
- Labels become addresses

Machine Implementation
- ALU, Control, Register File, ...
Instruction Usage

Instructions are stored in memory, encoded in binary.

A basic processor
  • fetches
  • decodes
  • executes
  one instruction at a time
MIPS Design Principles

Simplicity favors regularity
  • 32 bit instructions

Smaller is faster
  • Small register file

Make the common case fast
  • Include support for constants

Good design demands good compromises
  • Support for different type of interpretations/classes
Instruction Types

Arithmetic
  • add, subtract, shift left, shift right, multiply, divide

Memory
  • load value from memory to a register
  • store value to memory from a register

Control flow
  • unconditional jumps
  • conditional jumps (branches)
  • jump and link (subroutine call)

Many other instructions are possible
  • vector add/sub/mul/div, string operations
  • manipulate coprocessor
  • I/O
Instruction Set Architecture

The types of operations permissible in machine language define the ISA

- MIPS: load/store, arithmetic, control flow, ...
- VAX: load/store, arithmetic, control flow, strings, ...
- Cray: vector operations, ...

Two classes of ISAs

- Reduced Instruction Set Computers (RISC)
- Complex Instruction Set Computers (CISC)

We’ll study the MIPS ISA in this course
Instruction Set Architecture

Instruction Set Architecture (ISA)
- Different CPU architecture specifies different set of instructions. Intel x86, IBM PowerPC, Sun Sparc, MIPS, etc.

MIPS (RISC)
- ≈ 200 instructions, 32 bits each, 3 formats
- all operands in registers
- ≈ 1 addressing mode: Mem[reg + imm]

x86: Complex Instruction Set Computer (CISC)
- > 1000 instructions, 1 to 15 bytes each
- operands in special registers, general purpose registers, memory, on stack, ...
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  - e.g. Mem[segment + reg + reg*scale + offset]
Instructions

Load/store architecture
- Data must be in registers to be operated on
- Keeps hardware simple

Emphasis on efficient implementation

Integer data types:
- byte: 8 bits
- half-words: 16 bits
- words: 32 bits

MIPS supports signed and unsigned data types
# MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats

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

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>op</th>
<th>immediate (target address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Only 5 bits means that 32 offset. But need larger offset. So we go to having a 3rd type of instruction.

**MIPS Design Principles**

Simplicity favors regularity
- 32 bit instructions

Smaller is faster
- Small register file

Make the common case fast
- Include support for constants

Good design demands good compromises
- Support for different type of interpretations/classes of instructions
Takeaway

A MIPS processor and ISA (instruction set architecture) is an example of a Reduced Instruction Set Computers (RISC) where simplicity is key, thus enabling us to build it!!
Next Goal

How are instructions executed?
What is the general datapath to execute an instruction?
Instructions are stored in memory, encoded in binary.

A basic processor
- fetches
- decodes
- executes

one instruction at a time
Five Stages of MIPS Datapath

A Single cycle processor
Five Stages of MIPS datapath

Basic CPU execution loop
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

Instruction types/format
- Arithmetic/Register: addu $s0, $s2, $s3
- Arithmetic/Immediate: slti $s0, $s2, 4
- Memory: lw $s0, 20($s3)
- Control/Jump: j 0xdeadbeef
Stages of datapath (1/5)

Stage 1: Instruction Fetch
- Fetch 32-bit instruction from memory
  - Instruction cache or memory
- Increment PC accordingly
  - +4, byte addressing
  - +N
Stages of datapath (1/5)

A Single cycle processor
Stages of datapath (2/5)

Stage 2: Instruction Decode

- Gather data from the instruction
- Read opcode to determine instruction type and field length
- Read in data from register file
  - E.g. for addu, read two registers
  - E.g. for addi, read one register
  - E.g. for jal, read no registers
# Stages of datapath (2/5)

All MIPS instructions are 32 bits long, has 3 formats:

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>op rs rt rd shamt func</td>
<td>6 5 5 5 5 6</td>
</tr>
<tr>
<td>I-type</td>
<td>op rs rt immediate</td>
<td>6 5 5 16</td>
</tr>
<tr>
<td>J-type</td>
<td>op immediate (target address)</td>
<td>6 26</td>
</tr>
</tbody>
</table>

The R-type instruction is used for arithmetic and logical operations. The I-type instruction is used for load and store operations. The J-type instruction is used for jump operations.
Stages of datapath (2/5)

A Single cycle processor
Stages of datapath (3/5)

Stage 3: Execution (ALU)

- Useful work is done here (+, -, *, /), shift, logic operation, comparison (slt).
- Load/Store?
  - lw $t2, 32($t3)
  - Compute the address of the memory
A Single cycle processor
Stages of datapath (4/5)

Stage 4: Memory access
• Used by load and store instructions only
• Other instructions will skip this stage
Stages of datapath (4/5)

A Single cycle processor
Stages of datapath (5/5)

Stage 5:
- For instructions that need to write value to register
- Examples: arithmetic, logic, shift, etc., load
- Branches, jump??

New instruction address
If branch or jump

WriteBack from ALU or Memory

PC

Reg.
File
Takeaway

The datapath for a MIPS processor has five stages:
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

This five stage datapath is used to execute all MIPS instructions
Next Goal

Specific datapaths for MIPS Instructions
MIPS Instruction Types

Arithmetic/Logical
- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access
- load/store between registers and memory
- word, half-word and byte operations

Control flow
- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute
## MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats

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

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>immediate (target address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-type</td>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
## Arithmetic Instructions

```
00000001000001100010000000100110
```

### R-Type

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>func</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x21</td>
<td>ADDU rd, rs, rt</td>
<td>R[rd] = R[rs] + R[rt]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x23</td>
<td>SUBU rd, rs, rt</td>
<td>R[rd] = R[rs] - R[rt]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x25</td>
<td>OR rd, rs, rt</td>
<td>R[rd] = R[rs]</td>
<td>R[rt]</td>
<td></td>
</tr>
<tr>
<td><strong>0x0</strong></td>
<td><strong>0x26</strong></td>
<td>XOR rd, rs, rt</td>
<td>R[rd] = R[rs] ⊕ R[rt]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x27</td>
<td>NOR rd, rs rt</td>
<td>R[rd] = ~( R[rs]</td>
<td>R[rt] )</td>
<td></td>
</tr>
</tbody>
</table>

```
ex: r4 = r8 ⊕ r6  # XOR r4, r8, r6```

## Arithmetic Instructions: Shift

<table>
<thead>
<tr>
<th>op</th>
<th>func</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x0</td>
<td>SLL rd, rt, shamt</td>
<td>( R[rd] = R[rt] \ll \text{shamt} )</td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>SRL rd, rt, shamt</td>
<td>( R[rd] = R[rt] \ggg \text{shamt} ) (zero ext.)</td>
</tr>
<tr>
<td>0x0</td>
<td>0x3</td>
<td>SRA rd, rt, shamt</td>
<td>( R[rd] = R[rt] \gg \text{shamt} ) (sign ext.)</td>
</tr>
</tbody>
</table>

ex: \( r8 = r4 \times 64 \)  # SLL r8, r4, 6
\( r8 = r4 \ll 6 \)