Summary

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

- SRAM: caches
- DRAM: main memory
A Simple Processor

Chapters 2 and 5 in book

Instructions

- Programs are written in a high-level language
  - C, Java, Python, Miranda, …
  - Loops, control flow, variables

- Need translation to a lower-level computer understandable format
  - Processors operate on machine language
  - Assembler is human-readable machine language

```c
for(i = 0; i < 10; ++i)
    printf("go cornell cs");
```

```assembly
li r2, 10
li r1, 0
slt r3, r1, r2
bne ...
```

```
01001001000001010
01001000100000000
10001001100010010
```

© Kavita Bala, Computer Science, Cornell University
Basic Computer System

- 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

Instruction Usage

- Instructions are stored in memory, encoded in binary
- A basic processor
  - fetches
  - decodes
  - executes
  one instruction at a time
Instruction Types

- Arithmetic
  - add, subtract, shift left, shift right, multiply, divide
  - compare
- Control flow
  - unconditional jumps
  - conditional jumps (branches)
  - subroutine call and return
- Memory
  - load value from memory to a register
  - store value to memory from a register
- Many other instructions are possible
  - vector add/sub/mul/div, string operations, store internal state of processor, restore internal state of processor, manipulate coprocessor

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
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 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
Register file

- The MIPS register file
  - 32 32-bit registers
  - register 0 is permanently wired to 0
  - Write-Enable and RW determine which reg to modify
  - Two output ports A and B
  - RA and RB choose values read on outputs A and B
  - Writes occur on falling edge if WE is high

Memory

- 32-bit address
- 32-bit data
  - word = 32 bits
- 2-bit memory control input

© Kavita Bala, Computer Science, Cornell University
Instruction Fetch

- Read instruction from memory
- Calculate address of next instruction
- Fetch next instruction

Arithmetic Instructions

<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>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- if op == 0 && func == 0x21 ADD rd, rs, rt
  - R[rd] = R[rs] + R[rt] (unsigned) ADDU rd, rs, rt
- if op == 0 && func == 0x23 AND rd, rs, rt
  - R[rd] = R[rs] - R[rt] (unsigned) OR rd, rs, rt Nor rd, rs, rt
- if op == 0 && func == 0x25
  - R[rd] = R[rs] | R[rt]
Arithmetic Ops

- Memory
- Instruction (inst)
- Register file
- New PC
- Calculation
- Arithmetic Logic Unit (alu)

© Kavita Bala, Computer Science, Cornell University

Arithmetic Logic Unit

- Implements add, sub, or, and, shift-left, ...
- Operates on operands in parallel
- Control mux selects desired output

© Kavita Bala, Computer Science, Cornell University
Arithmetic Instructions (2)

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

• if op == 8
  – R[rd] = R[rs] + sign_extend(immediate)

• if op == 12
  – R[rd] = R[rs] & immediate

  ADDI rd, rs, val
  ADDIU rd, rs, val
  ANDI rd, rs, val
  ORI rd, rs, val

Sign Extension

• Often need to convert a small (8-bit or 16-bit) signed value to a larger (16-bit or 32-bit) signed value
  – “1” in 8 bits: 00000001
  – “1” in 16 bits: 0000000000000001
  – “-1” in 8 bits: 11111111
  – “-1” in 16 bits: 1111111111111111

• Conversion from small to larger numbers involves replicating the sign bit
Arithmetic Ops with Immediates

Summary

- With an ALU and fetch-decode unit, we have the equivalent of Babbage’s computation engine
  - Much faster, more reliable, no mechanical parts

- Next: control flow and memory operations
MIPS Instruction Types

• Arithmetic/Logical
  – three operands: result + two sources
  – operands: registers, 16-bit immediates
  – signed and unsigned versions

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

• Control flow
  – conditional branches: pc-relative addresses
  – jumps: fixed offsets

Memory Operations

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

• lb, lbu, lh, lhu, lw
• sb, sh, sw
• Examples  rs = r4 = addr of array; r3 = rd
  – lw r3, 0(r4)  int array[32]; x = array[0]
  – lw r3, 16(r4) int array[32]; x = array[4]
  – sw r3, 0(r4) array[0] = x;
Load

Store
Control Flow (Absolute Jump)

- \( j, \ jal \)
- Absolute addressing
- new PC = high 4 bits of current PC || target || 00
  - Cannot jump from 0xffff000000000000 to 0x0000100000000000
  - Better to make all instructions fit in 32 bits than to support really large absolute jumps
- Examples
  - \( j \ L01 \) goto L01

Absolute Jump

```
op | target
---|---
6 bits | 26 bits
```

© Kavita Bala, Computer Science, Cornell University