Review 1/3
- Apply Principle of Locality Recursively
- Reduce Miss Penalty? add a (L2) cache
- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings vs. tag/data in cache
- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB

Review 2/3
- Virtual Memory allows protected sharing of memory between processes with less swapping to disk, less fragmentation than always swap or base/bound
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
- TLB to reduce performance cost of VM
- Need more compact representation to reduce memory size cost of simple 1-level page table (especially 32-⇒ 64-bit address)

Review 3/3: Paging/Virtual Memory

User A:
Virtual Memory

User B:
Virtual Memory

Five Components of a Computer

Outline
- Datapath Walkthroughs
- Hardware Building Blocks
- ALU Design
- Full Adder
- Datapath utilization
The CPU

- **Processor** (CPU): the active part of the computer, which does all the work (data manipulation and decision-making)
- **Datapath**: portion of the processor which contains hardware necessary to perform all operations required by the computer (the brawn)
- **Control**: portion of the processor (also in hardware) which tells the datapath what needs to be done (the brain)

Stages of the Datapath (1/6)

- **Problem**: a single, atomic block which “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- **Solution**: break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others

Stages of the Datapath (2/6)

- There is a wide variety of MIPS instructions: so what general steps do they have in common?
- **Stage 1: Instruction Fetch**
  - no matter what the instruction, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)
  - also, this is where we **Increment PC** (that is, PC = PC + 4, to point to the next instruction: byte addressing so + 4)

Stages of the Datapath (3/6)

- **Stage 2: Instruction Decode**
  - upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
  - first, read the Opcode to determine instruction type and field lengths
  - second, read in data from all necessary registers
    - for add, read two registers
    - for addi, read one register
    - for jal, no reads necessary

Stages of the Datapath (4/6)

- **Stage 3: ALU (Arithmetic-Logic Unit)**
  - the real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (slt)
  - what about loads and stores?
    - lw $t0, 40($t1)
    - the address we are accessing in memory = the value in $t1 PLUS the value 40
    - so we do this addition in this stage

Stages of the Datapath (5/6)

- **Stage 4: Memory Access**
  - actually only the load and store instructions do anything during this stage; the others remain idle
  - since these instructions have a unique step, we need this extra stage to account for them
  - as a result of the cache system, this stage is expected to be just as fast (on average) as the others
Stages of the Datapath (6/6)

°Stage 5: Register Write
  • most instructions write the result of some computation into a register
  • examples: arithmetic, logical, shifts, loads, slt
  • what about stores, branches, jumps?
    - don’t write anything into a register at the end
    - these remain idle during this fifth stage

Datapath Walkthroughs (1/3)

°add $r3, $r1, $r2 # r3 = r1+r2
  • Stage 1: fetch this instruction, inc. PC
  • Stage 2: decode to find it’s an add, then read registers $r1 and $r2
  • Stage 3: add the two values retrieved in Stage 2
  • Stage 4: idle (nothing to write to memory)
  • Stage 5: write result of Stage 3 into register $r3

Example: ADD Instruction

Datapath Walkthroughs (2/3)

°slti $r3, $r1, 17
  • Stage 1: fetch this instruction, inc. PC
  • Stage 2: decode to find it’s an slti, then read register $r1
  • Stage 3: compare value retrieved in Stage 2 with the integer 17
  • Stage 4: go idle
  • Stage 5: write the result of Stage 3 in register $r3

Example: SLTI Instruction
**Datapath Walkthroughs (3/3)**

sw $r3, 17($r1)

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it’s a sw, then read registers $r1 and $r3
- Stage 3: add 17 to value in register $r4 (retrieved in Stage 2)
- Stage 4: write value in register $r3 (retrieved in Stage 2) into memory address computed in Stage 3
- Stage 5: go idle (nothing to write into a register)

**Example: SW Instruction**

![Example: SW Instruction diagram](image)

**Administrivia**

- Homework 8 (next week)
  - Want to fill in page tables to learn material, so easiest way is to turn in paper; no electronic submission
- Grading scale (same as Spring 99, Fall 99)
  - 95% A+, 90% A, 85% A-, 80% B+, 75% B, 70% B-, 65% C+, 60% C, 55% C-, 45% D

**Why Five Stages? (1/2)**

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does MIPS have five if instructions tend to go idle for at least one stage?
  - There is one instruction that uses all five stages: the load

**Why Five Stages? (2/2)**

lw $r3, 17($r1)

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it’s a lw, then read register $r1
- Stage 3: add 17 to value in register $r1 (retrieved in Stage 2)
- Stage 4: read value from memory address compute in Stage 3
- Stage 5: write value found in Stage 4 into register $r3
Example: LW Instruction

```
LW r3, 17(r1)
```

What Hardware Is Needed? (1/2)

° PC: a register which keeps track of VA of the next instruction
° General Purpose Registers
  • used in Stages 2 (Read) and 5 (Write)
  • we’re currently working with 32 of these
° Memory
  • used in Stages 1 (Fetch) and 4 (R/W)
  • cache system makes these two stages as fast as the others, on average

Datapath Summary

° Construct datapath based on register transfers required to perform instructions
° Control part causes the right transfers to happen

What Hardware Is Needed? (2/2)

° ALU
  • used in Stage 3
  • something that performs all necessary types of work: arithmetic, logicals, etc.
  • we’ll design this later
° Miscellaneous Registers
  • hold intermediate data, such as results in between stages, etc.
  • Note: Register is a general purpose term meaning something that stores bits.

Hardware Building Blocks (1/6)

° In reality, CPUs are built out of transistors and wires (plus resistors and capacitors).
° For this class, we’ll do design using gates and wires.
° Gate:
  • hardware unit that receives a certain number of inputs and produces one output: implements one of the basic logic functions
  • can be represented as a truth table
  • actually implemented in transistors

Hardware Building Blocks (2/6)

**AND Gate**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>called a &quot;truth table&quot;</td>
</tr>
<tr>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td></td>
</tr>
</tbody>
</table>
**Hardware Building Blocks (3/6)**

**OR Gate**

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

**Definition**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

° We can have more inputs:
- \( C = 1 \) if and only if ANY input is 1

**Hardware Building Blocks (4/6)**

**Inverter**

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

**Definition**

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

° In this case, there is always exactly one input and one output.
° Note: Inverter is usually drawn as just a bubble, without the triangle.

**Hardware Building Blocks (5/6)**

**Multiplexor (MUX)**

<table>
<thead>
<tr>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>

**Definition**

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Hardware Building Blocks (6/6)**

° General Muxes:
- have control bits and data bits
- control bits select which data bit will pass through: all others are blocked
- in general:
  - 1 control bit selects between 2 data bits,
  - 2 control bits select between 4 data bits,
  - 3 control bits select between 8 data bits,
  - in general, \( 2^n \) control bits select between \( 2^n \) data bits
- so we can build a mux of any size to serve our purpose

**Registers**

° General Definition of a Register
- a place where we can store one or more bits for future retrieval
- since registers are in hardware, they can be designed using gates and wires
° In MIPS, we have:
- 32 general purpose registers used by programs for computations
- registers in the datapath used to store data between Stages
- a few more (such as PC, hi, lo)

**ALU Design Philosophies (1/4)**

° Fact: All basic hardware building blocks accept individual bits at inputs and output individual bit.
° Fact: The MIPS ALU (generally) needs to work with 32-bit registers.
° Design Philosophy #1: For simplicity, build 32 separate one-bit ALUs, and then figure out what needs to be done to connect them.
ALU Design Philosophies (2/4)

- Fact: ALU needs to perform a **WIDE** variety of tasks: add, subtract, multiply, divide, shift, compare, etc.

- Design Philosophy #2:
  - Build separate hardware blocks for each necessary task.
  - When inputs come in, perform all possible operations in parallel.
  - Use a mux to choose which operation is actually desired.
  - Note: Since everything’s done in parallel, no time is lost.

ALU Design Philosophies (3/4)

- Consequence of Design Philosophy #2:
  - New operations can be added to the ALU just by adding new data lines into the muxes and informing Control of the change.
  - This means that new instructions can be added to the system without changing everything: just a small portion of the ALU.

ALU Design Philosophy (4/4)

- There will be more places in the design where we need to make a decision:
  - so feel free to add a mux whenever necessary
  - assume that control signals for all muxes can be handled by the Control

In the Beginning

- The ALU consists of 32 muxes (one for each necessary output bit).

Logical Instructions (1/2)

- AND instruction
  - one of data lines should be a simple AND gate

- OR instruction
  - another data line should be a simple OR gate

One-Bit Full Adder (1/3)

- Example Binary Addition:
  - a: 0 0 1 1
  - b: 0 1 0 1
  - Sum: 1 0 0 0

- Thus for any bit of addition:
  - The inputs are a_i, b_i, CarryIn_i
  - The outputs are Sum_i, CarryOut_i

- Note: CarryIn_{i+1} = CarryOut_i
One-Bit Full Adder (2/3)

Symbol | Definition
---|---
A | B | CarryIn | CarryOut | Sum
0 | 0 | 0 | 0 | 0
0 | 1 | 0 | 0 | 1
0 | 1 | 1 | 1 | 0
1 | 0 | 0 | 0 | 1
1 | 0 | 1 | 1 | 0
1 | 1 | 0 | 1 | 0
1 | 1 | 1 | 1 | 1

Add 1-bit Adder to 1-bit ALU

Definition

| Op | C |
---|---|
0 | A and B |
1 | A or B |
2 | A + B + CarryIn |

°Now connect 32 1-bit ALUs together

One-Bit Full Adder (3/3)

°To create one-bit full adder:
  • implement gates for Sum
  • implement gates for CarryOut
  • connect all inputs with same name
  • the symbol for one-bit full adder now represents this jumble of gates and wires (simplifies schematics)

Constructing Hardware to Match Definition

°Given any table of binary inputs for a binary output, programs can automatically connect a minimal number of AND gates, OR gates, and Inverters to produce the desired function

°Such programs generically called "Computer Aided Design", or CAD

Example: HW gates for CarryOut

°Values of Inputs when CarryOut is 1:

| A | B | CarryIn |
---|---|---|
0 | 1 | 1 |
1 | 0 | 1 |
1 | 1 | 0 |
1 | 1 | 1 |

°Gates for CarryOut signal:

°Gates for Sum left as exercise to Reader

Things to Remember (1/3)

°Datapath is the hardware that performs operations necessary to execute programs.

°Control instructs datapath on what to do next.

°Datapath needs:
  • access to storage (general purpose registers and memory)
  • computational ability (ALU)
  • helper hardware (local registers and PC)
Things to Remember (2/3)

° Five stages of datapath (executing an instruction):
  1. Instruction Fetch (Increment PC)
  2. Instruction Decode (Read Registers)
  3. ALU (Computation)
  4. Memory Access
  5. Write to Registers

° ALL instructions must go through ALL five stages.

° Datapath designed in hardware.

Things to Remember (3/3)

° When inputs come into ALU stage, all possible calculations/operations are performed on them.

° One big mux then chooses which operation is actually desired.

° New functionality can be added simply by modifying the existing ALU (adding a new data line to the mux, if necessary)

° Computer Aided Design can create gates to implement function defined in any truth table