Designing a Single Cycle Datapath

Outline of Today’s Lecture

- Homework #3 Due March 2
- Projects due April 20 (tell me which project)
- Reading Ch 5.1-5.3
- Where are we with respect to the BIG picture?
- The Steps of Designing a Processor
- Datapath and timing for Reg-Reg Operations
- Datapath for Logical Operations with Immediate
- Datapath for Load and Store Operations
- Datapath for Branch and Jump Operations
Review: Integer Multiplication & Division

**Multiplication**
- Series of Shift and Add

**Booth’s Algorithm**
- (Current, Previous) bits of Multiplier:
  - 0,0: middle of string of 0s; do nothing
  - 0,1: end of a string of 1s; add multiplicand
  - 1,0: start of string of 1s; subtract multiplicand
  - 1,1: middle of string of 1s; do nothing
- Shift Product/Multiplier right 1 bit (as before)

**Division**
- Series of Shift and Subtract

Review: FP Addition

- Example:
  - $1.610 \times 10^{-1} + 9.999 \times 10^1$
- Step 1:
  - Align decimal point: $0.016 \times 10^1 + 9.999 \times 10^1$
- Step 2:
  - Add: $10.015 \times 10^1$
- Step 3:
  - Normalize: $1.0015 \times 10^2$
- Step 4:
  - Round: $1.002 \times 10^2$
- May need to repeat steps 3 and 4 if result not normal after rounding. (renormalization)
Review: FP Multiplication

1. Add biased exponents, subtract bias
2. Multiply significands
3. Normalize product
4. Round significand
5. Compute sign of product

° \(.5 \times - .75 \Rightarrow 1.000 \times 2^{-1} \times 1.100 \times 2^{-1}\)

Example FP Multiply (Corrected)

° \(.5 \times - .75 \Rightarrow 1.000 \times 2^{-1} \times 1.100 \times 2^{-1}\)
° With Bias \(1.000 \times 2^{126} \times 1.100 \times 2^{126}\)
1. \(126 + 126 - 127 = 125\)
2. 
\[
\begin{array}{l}
1.000 \\
1.100 \\
0000 \\
0000 \\
1000 \\
1000 \\
1.1000000 \\
\end{array}
\]
3. Normalize product: \(1.1000000 \times 2^{125}\)
4. Round: \(1.100 \times 2^{125}\)
5. Compute Sign: different so result is neg 
- \(1.100 \times 2^{125}\) (biased) = - \(1.100 \times 2^{-2} = -.375\)
Accuracy

- Is \((x+y)+z = x + (y+z)\)?
- Computer numbers have limited size \(\Rightarrow\) limited precision.
- Rounding Errors

- Example:
  \[2.56 \times 10^0 + 2.34 \times 10^2,\] using 3 significant digits

- Align decimal points (exponents, shift smaller)
  \[
  \begin{align*}
    2.34 & \times 10^2 \\
    0.0256 & \times 10^2 \\
    2.36 & \times 10^2
  \end{align*}
  \]

Rounding

- Rounding with Guard & Round bits

- Example:
  \[2.56 \times 10^0 + 2.34 \times 10^2,\] using 3 significant digits

- Align decimal points (exponents, shift smaller)
  \[
  \begin{align*}
    2.34 & \\
    0.0256 & \quad \text{Guard = 5, Round = 6} \\
    2.3656 &
  \end{align*}
  \]

- Round: \(2.37 \times 10^2\)

- Without guard & round bits, result = \(2.36 \times 10^2\)

- Error of 1 Unit in the least significant position

- Why 2 bits?
  - Product could have leading 0, so shift left when normalizing
Arithmetic Exceptions

- Conditions
  - Overflow
  - Underflow
  - Division by zero

- Special floating point values
  - +infinity (e.g. 1/0)
  - -infinity (e.g. -1/0)
  - NaN (Not A Number) (e.g. 0/0, infinity/infinity, sq. root of -1)

What is Computer Architecture?

- Coordination of levels of abstraction

  ![Diagram of computer architecture](image)

  - Under a set of rapidly changing technology Forces

Software

Interface Between HW and SW

Instruction Set Architecture, Memory, I/O

Hardware
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

![Diagram showing the five components of a computer: Processor (Control, Datapath), Memory, Input, Output]

Today’s Topic: Datapath Design

Datapath Design

- How do we build hardware to implement the MIPS instructions?
- Add, LW, SW, Beq, Jump
### The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:
  - **R-type**
    - `op` (6 bits)
    - `rs` (5 bits)
    - `rt` (5 bits)
    - `rd` (5 bits)
    - `shamt` (5 bits)
    - `funct` (6 bits)
  - **I-type**
    - `op` (6 bits)
    - `rs` (5 bits)
    - `rt` (5 bits)
    - `immediate` (16 bits)
  - **J-type**
    - `op` (6 bits)
    - `target address` (26 bits)

- The different fields are:
  - `op`: operation of the instruction
  - `rs`, `rt`, `rd`: the source and destination register specifiers
  - `shamt`: shift amount
  - `funct`: selects the variant of the operation in the “op” field
  - `address / immediate`: address offset or immediate value
  - `target address`: target address of the jump instruction

### The MIPS Subset (We can't implement them all!)

- **ADD and subtract**
  - `add rd, rs, rt`
  - `sub rd, rs, rt`
- **OR Immediate**
  - `ori rt, rs, imm16`
- **LOAD and STORE**
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`
- **BRANCH**
  - `beq rs, rt, imm16`
- **JUMP**
  - `j target`
The Hardware “Program”

How do I build the hardware to implement the MIPS instructions and their sequencing?

Combinational Logic Elements (Basic Building Blocks)

° Adder

° MUX

° ALU
**Storage Element: Register (Basic Building Block)**

- **Register**
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - negated (0): Data Out will not change
    - asserted (1): Data Out will become the same as Data In.

**Storage Element: Register File**

- **Register File consists of 32 registers:**
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW

- **Register is selected by:**
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable is 1

- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after “access time.”
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out

- Memory word is selected by:
  - Write Enable = 0: Address selects the word to put on the Data Out bus
  - Write Enable = 1: Address selects the memory word to be written via the Data In bus

- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after “access time.”

An Abstract View of the Implementation

- Instruction Address
- Ideal Instruction Memory
- Ideal Data Memory
- 32 32-bit Registers
- Clk
- Data Out
- Data Address
**Clocking Methodology**

- All storage elements are clocked by the same clock edge
- Cycle Time $\geq$ CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- Longest delay path = critical path

**An Abstract View of the Critical Path**

- Register file and ideal memory:
  - The CLK input is a factor ONLY during write operation
  - During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”
The Steps of Designing a Processor

° Instruction Set Architecture => Register Transfer Language
° Register Transfer Language =>
  • Datapath components
  • Datapath interconnect
° Datapath components => Control signals
° Control signals => Control logic

Overview of the Instruction Fetch Unit

° The common RTL operations
  • Fetch the Instruction: mem[PC]
  • Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump: PC <- “something else”
RTL: The ADD Instruction

° add rd, rs, rt

- mem[PC] Fetch the instruction from memory
- PC <- PC + 4 Calculate the next instruction's address

RTL: The Load Instruction

° lw rt, rs, imm16

- mem[PC] Fetch the instruction from memory
- Address <- R[rs] + SignExt(imm16) Calculate the memory address
- R[rt] <- Mem[Address] Load the data into the register
- PC <- PC + 4 Calculate the next instruction's address
RTL: The ADD Instruction

- **add rd, rs, rt**
  - `mem[PC]` Fetch the instruction from memory
  - `PC <- PC + 4` Calculate the next instruction's address

RTL: The Subtract Instruction

- **sub rd, rs, rt**
  - `mem[PC]` Fetch the instruction from memory
  - `PC <- PC + 4` Calculate the next instruction's address
Datapath for Register-Register Operations

° $R[rd] \leftarrow R[rs] \text{ op } R[rt]$

- $Ra$, $Rb$, and $Rw$ comes from instruction’s $rs$, $rt$, and $rd$ fields
- $ALUctr$ and $RegWr$: control logic after decoding the instruction fields: $op$ and $func$

### Datapath Diagram

**Legend:***
- **Rw**: Write back register
- **Rs**, **Rt**, **Rd**: Source and destination registers
- **busA**, **busB**: Bus signals
- **RegWr**: Register write signal
- **Clk**: Clock signal
- **ALUctr**: ALU control

### Instruction Fetch Timing

- **Inst fetch**: Instructions are fetched from memory
- **Decode**: Instructions are decoded
- **Opr. fetch**: Operands are fetched from registers
- **Execute**: Instructions are executed
- **Write Back**: Results are written back

### Table: Register Write Timing

<table>
<thead>
<tr>
<th>Field</th>
<th>Old Value</th>
<th>New Value</th>
<th>Delay</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs, Rt, Rd, Op, Func</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUctr</td>
<td>Old</td>
<td>New</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegWr</td>
<td>Old</td>
<td>New</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busA, B</td>
<td>Old</td>
<td>New</td>
<td></td>
<td></td>
</tr>
<tr>
<td>busW</td>
<td>Old</td>
<td>New</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram Details

- **32-bit Registers**
- **ALU Delay**
- **Register File Access Time**
- **Instruction Memory Access Time**
- **Delay through Control Logic**

**Diagrams:**
- Datapath for Register-Register Operations
- Register-Register Timing
**RTL: The OR Immediate Instruction**

- **ori rt, rs, imm16**
  - **mem[PC]**: Fetch the instruction from memory
  - **R[rt] <- R[rs] or ZeroExt(imm16)**: The OR operation
  - **PC <- PC + 4**: Calculate the next instruction’s address

### Datapath for Logical Operations with Immediate

  - **Example: ori rt, rs, imm16**

---

**Diagram of Datapath**

- **Rw, Ra, Rb**
- **RegWr**
- **RegDst**
- **busA, busB**
- **Clk**
- **imm16**
- **Mux**
- **ALUctr**
- **ALUSrc**
- **Rd**
- **Rt**
- **Don’t Care (Rt)**
- **32-bit Registers**
- **Result**
**RTL: The Load Instruction**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>rt, rs, imm16</td>
<td>6 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

- **mem[PC]**
  Fetch the instruction from memory

- **Address <- R[rs] + SignExt(imm16)**
  Calculate the memory address

- **R[rt] <- Mem[Address]**
  Load the data into the register

- **PC <- PC + 4**
  Calculate the next instruction’s address

**Datapath for Load Operations**

  Example: lw rt, rs, imm16

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</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>
**RTL: The Store Instruction**

- **sw rt, rs, imm16**
  - `mem[PC]` Fetch the instruction from memory
  - `Address <- R[rs] + SignExt(imm16)` Calculate the memory address
  - `Mem[Address] <- R[rt]` Store the register into memory
  - `PC <- PC + 4` Calculate the next instruction's address

---

**Datapath for Store Operations**

- `Mem[R[rs] + SignExt(imm16)] <- R[rt]` Example: `sw rt, rs, imm16`
**RTL: The Branch Instruction**

beq  rs, rt, imm16

- **mem[PC]** Fetch the instruction from memory
- **Cond <- R[rs] - R[rt]** Calculate the branch condition
- **if (COND eq 0)** Calculate the next instruction's address
  - \( PC <- PC + 4 + (\text{SignExt}(\text{imm16}) \times 4) \)
- **else**
  - \( PC <- PC + 4 \)

**Datapath for Branch Operations**

We need to compare Rs and Rt!

beq  rs, rt, imm16
**Binary Arithmetic for the Next Address**

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: \( \text{PC}<31:0> = \text{PC}<31:0> + 4 \)
  - Branch operation: \( \text{PC}<31:0> = \text{PC}<31:0> + 4 + \text{SignExt}[\text{Imm16}] \times 4 \)

- The magic number “4” always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long

- In other words:
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs

- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: \( \text{PC}<31:2> = \text{PC}<31:2> + 1 \)
  - Branch operation: \( \text{PC}<31:2> = \text{PC}<31:2> + 1 + \text{SignExt}[\text{Imm16}] \)
  - In either case: Instruction-Memory-Address = PC<31:2> concat “00”

---

**Next Address Logic: Expensive and Fast Solution**

- Using a 30-bit PC:
  - Sequential operation: \( \text{PC}<31:2> = \text{PC}<31:2> + 1 \)
  - Branch operation: \( \text{PC}<31:2> = \text{PC}<31:2> + 1 + \text{SignExt}[\text{Imm16}] \)
  - In either case: Instruction-Memory-Address = PC<31:2> concat “00”
RTL: The Jump Instruction

- Fetch the instruction from memory

- \( \text{PC} \leftarrow \text{PC} + 4 \ll 28 \) concat target\( \ll 25 \) concat \(00\)
  Calculate the next instruction's address
Instruction Fetch Unit

\[ \text{target = PC} <31:2> \leftarrow \text{PC} + 4 <31:28> \text{ concat target}<25:0> \]

Putting it All Together: A Single Cycle Datapath

We have everything except control signals.