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

- $.5 \times .75 \Rightarrow 1.000x2^{-1} \times 1.100x2^{-1}$

Example FP Multiply (Corrected)

- $.5 \times .75 \Rightarrow 1.000x2^{-1} \times 1.100x2^{-1}$
  - With Bias $1.000x2^{126} \times 1.100x2^{126}$
  1. $126+126-127 = 125$
  2. 
    
    1.000
    1.100
    0000
    1000
    1000
    1.000000
  3. Normalize product: $1.000000x2^{125}$
  4. Round: $1.000x2^{125}$
  5. Compute Sign: different so result is neg

- $-1.100x2^{-10}$ (biased) $= -1.100x2^{10} = -.375$
Accuracy

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

- Example:
  \[2.56 \times 10^2 + 2.34 \times 10^2, \text{ using 3 significant digits}\]
- Align decimal points (exponents, shift smaller)
  \[2.34 \times 10^2, 0.0256 \times 10^2, 2.36 \times 10^2\]

Rounding

- Rounding with Guard & Round bits
- Example:
  \[2.56 \times 10^2 + 2.34 \times 10^2, \text{ using 3 significant digits}\]
- Align decimal points (exponents, shift smaller)
  \[2.34, 0.0256, 2.3656\]
- Guard = 5, Round = 6
- 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
  - \(+\text{infinity} \) (e.g. \(1/0\))
  - \(-\text{infinity} \) (e.g. \(-1/0\))
  - \(\text{NaN} \) (Not A Number) (e.g. \(0/0, \text{infinity/infinity}, \text{sq. root of -1}\))

What is Computer Architecture?

- Coordination of levels of abstraction

- Under a set of rapidly changing technology Forces

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
- Instruction Set Architecture, Memory, I/O
- Interface Between HW and SW

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
  - I-type
  - J-type

- 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 Enables:
    - 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

Clocking Methodology

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

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: \texttt{mem[PC]}
  - Update the program counter:
    - Sequential Code: \texttt{PC <- PC + 4}
    - Branch and Jump: \texttt{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

**Datapath for Register-Register Operations**

- R[rd] <- R[rs] op R[rt]
- Rs, Rs, and Rd come from instruction’s rs, rt, and rd fields
- ALUctr and RegWr: control logic after decoding the instruction fields: op and func

**Register-Register Timing**

- Inst fetch
- Decode
- Opr. fetch
- Execute
- Write Back
- Delay through Control Logic
- New Value
- Register File Access Time
- ALU Delay
- New Value
- Register Write
- New Value
- Result

Example: add rd, rs, rt

- Ra, Rb, and Rw comes from instruction’s rs, rt, and rd fields
RTL: The OR Immediate Instruction

```plaintext
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

```plaintext
Example: ori rt, rs, imm16
```

RTL: The Load Instruction

```plaintext
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
```

Datapath for Load Operations

```plaintext
Example: lw rt, rs, imm16
```

RTL: The Store Instruction

```plaintext
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

```plaintext
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 + (SignExt[imm16] x 4)
    - else
    - PC ← PC + 4

**Datapath for Branch Operations**

- **beq rs, rt, imm16**
  - We need to compare Rs and Rt!

**Binary Arithmetic for the Next Address**

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[imm16] * 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: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[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: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[imm16]
  - In either case: Instruction-Memory-Address = PC<31:2> concat "00"

**RTL: The Jump Instruction**

- **j target**
  - mem[PC] Fetch the instruction from memory
  - PC ← PC + 4<31:28> concat target<25:0> concat <00>
  - Calculate the next instruction’s address
Putting it All Together: A Single Cycle Datapath

We have everything except control signals.

Instruction Fetch Unit

Putting it All Together: A Single Cycle Datapath

We have everything except control signals.