Another shutout for Cal!

Unbelievable! The #4 Bears were dominant in beating ASU 27-0. JJ Arrington shatters Cal records w/ his 7th-straight 100yd game, becoming the fastest Cal player ever to reach 1,000 yds. It’s ASU’s 1st shutout loss in 9 yrs & our first time in the top 5 in 52 years!!

OU next Sat...
Why is it “memArray[address[9:2]]”?

• Our memory is always byte-addressed
  • We can lb from 0x0, 0x1, 0x2, 0x3, …

• lw only reads word-aligned requests
  • We only call lw with 0x0, 0x4, 0x8, 0xC, …
  • I.e., the last two bits are always 0

• memArray is a word wide and $2^8$ deep
  • reg [31:0] memArray [0:256-1];
  • Size = 4 Bytes/row * 256 rows = 1024 B
  • If we’re simulating lw/sw, we R/W words
  • What bits select the first 256 words? [9:2]!

• 1st word = 0x0 = 0b000 = memArray[0];
• 2nd word = 0x4 = 0b100 = memArray[1], etc.
How to Design a Processor: step-by-step

1. Analyze instruction set architecture (ISA) => datapath requirements
   - meaning of each instruction is given by the register transfers
   - datapath must include storage element for ISA registers
   - datapath must support each register transfer

2. Select set of datapath components and establish clocking methodology

3. Assemble datapath meeting requirements

4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.

5. Assemble the control logic (hard part!)
Storage Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - negated (or deasserted) (0): Data Out will not change
  - asserted (1): Data Out will become Data In
Verilog 32-bit Register for MIPS Interpreter

// Behavioral model of 32-bit Register:  
// positive edge-triggered,  
// synchronous active-high reset.
module reg32 (CLK,Q,D,wEnb);
  input  CLK, wEnb;
  input  [31:0] D;
  output [31:0] Q;
  reg [31:0] Q;

  always @ (posedge CLK)
    if (wEnb)
      Q = D;
endmodule  // reg32
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 (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) 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.”
Verilog Register File for MIPS Interpreter (1/3)

// Behavioral model of register file:
// 32-bit wide, 32 words deep,
// two asynchronous read-ports,
// one synchronous write-port.
// Dump register file contents to
// console on pos edge of dump signal.
module regFile (CLK, wEnb, DMP, writeReg, writeD, readReg1, readD1, readReg2, readD2);
    input CLK, wEnb, DMP;
    input [4:0] writeReg, readReg1, readReg2;
    input [31:0] writeD;
    output [31:0] readD1, readD2;
    reg [31:0] readD1, readD2;
    reg [31:0] array [0:31];
    reg dirty1, dirty2;
    integer i;

• 3 5-bit fields to select registers: 1 write register, 2 read register
always @ (posedge CLK)
  if (wEnb)
    if (writeReg!=5'h0) // why?
      begin
        array[writeReg] = writeD;
        dirty1=1'b1;
        dirty2=1'b1;
      end
  always @ (readReg1 or dirty1)
    begin
      readD1 = array[readReg1];
      dirty1=0;
    end
Step 3: Assemble DataPath meeting requirements

- Register Transfer Requirements ➞ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
3a: 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”
3b: Add & Subtract

- $R[rd] = R[rs] \text{ op } R[rt]$  
  Ex.: $\text{addU } rd, rs, rt$

- $Ra, Rb, \text{ and } Rw$ come from instruction’s $Rs, Rt, \text{ and } Rd$ fields

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</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>

- $\text{ALUctr}$ and $\text{RegWr}$: control logic after decoding the instruction

- Already defined register file, ALU
Clocking Methodology

- Storage elements clocked by same edge
- Being physical devices, flip-flops (FF) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF, and we have the usual clock-to-Q delay
- “Critical path” (longest path through logic) determines length of clock period
Register-Register Timing: One complete cycle

- **Clk**: Clock signal.
- **PC**: Program Counter.
- **Rs, Rt, Rd**: Register File.
- **Op, Func**: Instructionopcode.
- **ALUctr**: ALU Control unit.
- **RegWr**: Register Write.
- **busA, B**: Bus signals for data.
- **busW**: Bus for result.

**Timing Phases**:

1. **Instruction Memory Access Time**: PC updates with new value.
2. **Delay through Control Logic**: Rs, Rt, Rd, Op, Func.
3. **Register File Access Time**: Rs, Rt, Rd.
4. **ALU Delay**: ALU operation.
5. **Register Write**: RWr occurs here.

**Diagram**:

- 32-bit Registers
- ALU for 32-bit operations
- Connections to buses and control signals.
3c: Logical Operations with Immediate

- \( R[rt] = R[rs] \text{ op ZeroExt}[imm_{16}] \)

What about \( Rt \) register read??

Already defined 32-bit MUX; Zero Ext?
3d: Load Operations

- \( R[rt] = \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)

Example: \( \text{lw } rt, rs, \text{imm16} \)

<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>

**Diagram:***

- \( R[rt] = \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)
- Example: \( \text{lw } rt, rs, \text{imm16} \)
- **Datapath:**
  - ALU
  - Memory
  - Registers
  - Extender
  - Muxes
  - Clk
  - Bus signals
  - **Verilog II:**
    - 16-bit data paths
    - Control signals
3e: Store Operations

  Ex.: sw rt, rs, imm16

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
<td></td>
</tr>
</tbody>
</table>
```

Example: Store operation

```
sw rt, rs, imm16
```

Diagram showing the datapath with registers, ALU, memory, and various control signals.
3f: The Branch Instruction

- **beq rs, rt, imm16**
  - mem[PC] Fetch the instruction from memory
  - Equal = R[rs] == R[rt] Calculate branch condition
  - if (Equal) 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`

Datapath generates condition (equal)

### Opcode Table

<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>

### Diagram

- **Inst Address**
- **Cond**
- **Already MUX, adder, sign extend, zero**
Putting it All Together: A Single Cycle Datapath

- Inst Memory Addr
- Instruction<31:0>
- Rs ▲ 21:25
- Rt ▲ 16:20
- Rd ▲ 11:15
- Imm16 ▲ 0:15

- nPC sel
- RegDst
- Rd ▲ 1
- Rt ▲ 0
- Rs 5
- Rt 5
- MemWr
- ALUctr
- MemWr MemtoReg

- PC Ext
- Adder
- Mux
- busW
- Clk 0
- 32
- 32
- 4
- imm16
- 16
- ExtOp
- ALUSrc

- 32 32-bit Registers
- PC
- Adder
- Mux
- Extender
- WrEnAdr
- Data In
- Memory
- Data
- Clk
- 0
- 1
An Abstract View of the Implementation
Suppose we’re writing a MIPS interpreter in Verilog. Which sequence below is best organization for the interpreter?

A. repeat loop that fetches instructions
B. while loop that fetches instructions
C. Decodes instructions using case statement
D. Decodes instr. using chained if statements
E. Executes each instruction
F. Increments PC by 4

1: ACEF
2: ADEF
3: AECF
4: AEDF
5: BCEF
6: BDEDF
7: BECF
8: BEDDF
9: EF
0: FAE
Summary: Single cycle datapath

° 5 steps to design a processor
  • 1. Analyze instruction set => datapath requirements
  • 2. Select set of datapath components & establish clock methodology
  • 3. Assemble datapath meeting the requirements
  • 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  • 5. Assemble the control logic

° Control is the hard part
° Next time!
Dwarfing the importance of this lecture...

...is the importance that tomorrow you get out and VOTE!