Designing A MIPS Single Cycle Datapath (I)

Dr. David R. Surma

Department of Computer and Information Sciences

Adapted from Morgan Kaufmann, Dr. L. Zhang and others
We're ready to look at an implementation of the MIPS

Simplified to contain only:
- memory-reference instructions: \texttt{lw}, \texttt{sw}
- arithmetic-logical instructions: \texttt{add}, \texttt{sub}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{nor}, \texttt{slt}
- arithmetic-logical immediate instructions: \texttt{addi}, \texttt{andi}, \texttt{ori}, \texttt{xori}, \texttt{slti}
- control flow instructions: \texttt{beq}, \texttt{bne}, \texttt{j}, \texttt{(jr, jal)}

Generic implementation:
- use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
- decode the instruction (and read registers)
- execute the instruction
  - actually include stages 3, 4, and 5 discussed in our previous lecture
Generic Steps of Datapath

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Reg. Write
Abstract Implementation View

- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

- Single cycle operation

- Split memory (Harvard) model - one memory for instructions and one for data
Combinational Logic Elements (Building Blocks)

- **Adder**
  - Inputs: A (32), B (32)
  - Outputs: Sum (32), CarryOut
  - CarryIn

- **MUX**
  - Inputs: A (32), B (32), Select
  - Outputs: Y (32)

- **ALU**
  - Inputs: A (32), B (32), OP
  - Output: Result (32)
Combinational Logic vs. Sequential Logic

- **Combinational logic:**
  - the output of the logic device is dependent only on the present inputs to the device.
  - Examples: adder, multiplex, decoder, ALU (without considering the registers)

- **Sequential logic**
  - the output of the logic device is dependent **not only** on the present inputs to the device, **but also** on past inputs;
  - i.e., the output of a sequential logic device depends on its present internal state and present inputs.
  - It has some kind of **memory** of at least part of its “history”
  - It contains **state element(s)**
  - Examples: latches, flip flops, registers, memory, counters…
The D Latch and the D Flip Flop

The truth table (state table) of D Flip Flop/D Latch

<table>
<thead>
<tr>
<th>Preset</th>
<th>Clear</th>
<th>Clock</th>
<th>D</th>
<th>Q</th>
<th>\overline{Q}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>↑ or 1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>↑ or 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>\overline{Q}_0</td>
<td>\overline{Q}_0</td>
</tr>
</tbody>
</table>

The Timing Diagram of D Latch and D Flip Flop

D Latch

Level sensitive

D Flip Flop

Edge-triggered

The D latch

(b) The D flip flop
Exercise

- Complete the timing diagram

Clock

D

D Latch

Q (Initial state: 0)

D Flip Flop

Q (Initial state: 0)
Latches vs Flipflops

- Output is equal to the stored value inside the element
- Change of state (value) is based on the clock
  - **Latches**: output changes whenever the inputs change and the clock is asserted (level sensitive methodology)
  - **Flip-flops**: output changes only on a clock edge (edge-triggered methodology)

- There are other type of flip flops: **RS, JK, T**. However, D type is the only type that we need in this class.
Clocking Methodologies

- Clocking methodology defines when signals (data) can be read and when they can be written
  - For edge-triggered clocking, when used for data writing, the clock edge acts a sampling signal causing the value of the data input to a state element to be sampled and stored in the state element.

- State element design choices
  - level sensitive latch
  - edge-triggered flipflops

- Clock rate = 1/(clock cycle)
  - e.g., 10 nsec clock cycle = 100 MHz clock rate
  - 1 nsec clock cycle = 1 GHz clock rate
State Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input

- RegWrite:
  - negated (or deasserted) (0):
    Data Out will not change
  - asserted (1):
    Data Out will become Data In on positive edge of clock
State 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 RegWrite 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.”
Register File – Read Ports Implementation

- **RegWrite**: RW, RA, RB
- **busW**: 32
- **Clk**: 5
- **busA**: 32
- **busB**: 32

### Registers

- Register 0
- Register 1
- ... (ellipses)
- Register \( n - 2 \)
- Register \( n - 1 \)

###mux

- **Read data 1**
- **Read data 2**
Register File – Write Port Implementation

- Register number
- Register data
- Write
- n-to-2^n decoder
- RegWrite
- RWRA RB
- busW
- Clk
- busA 32
- busB 32

32 32-bit Registers
State Element: Idealized Memory

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

- Memory word is selected by:
  - MemRead = 1: address selects the word to put on Data Out
  - MemWrite = 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 & MemRead (asserted) $\Rightarrow$ Data Out valid after “access time.”
Our Implementation

- An edge-triggered methodology
- Typical execution
  - read contents of some state elements
  - send values through some combinational logic
  - write results to one or more state elements

- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs
Fetching Instructions

- Fetching instructions involves:
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction

- PC is updated every clock cycle, so it does not need an explicit write control signal
- Instruction Memory is read every clock cycle, so it doesn’t need an explicit read control signal
Decoding Instructions

- Decoding instructions involves sending the fetched instruction’s opcode and function field bits to the control unit.

- reading two values from the Register File
  - Register File addresses are contained in the instruction.
Reading Registers “Just in Case”

- Note that both RegFile read ports are active for all instructions during the Decode cycle using the rs and rt instruction field addresses.
  - Since haven’t decoded the instruction yet, don’t know what the instruction is!
  - *Just in case* the instruction uses values from the RegFile do “work ahead” by reading the two source operands.

Which instructions *do* make use of the RegFile values?

- Also, all instructions (except *Jump* instructions) use the ALU *after* reading the registers.

Executing R Format Operations

- R format operations (add, sub, slt, and, or) perform operation (op and funct) on values in rs and rt.
- Store the result back into the Register File (into location rd).

Note that Register File is not written every cycle (e.g. sw), so we need an explicit write control signal for the Register File.
Consider the `slt` Instruction

- Remember the R format instruction `slt`

  
  ```
  slt $t0, $s0, $s1  # if $s0 < $s1
                    #  then $t0 = 1
                    #  else $t0 = 0
  ```

- Where does the 1 (or 0) come from to store into $t0 in the Register File at the end of the execute cycle?

![Diagram of ALU and Register File operations](diagram.png)
Executing Load and Store Operations

- Load and store operations have to compute a memory address by adding the base register (in rs) to the 16-bit signed offset field in the instruction
  - base register was read from the Register File during decode
  - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value

- store value, read from the Register File during decode, must be written to the Data Memory

- load value, read from the Data Memory, must be stored in the Register File
Executing Load and Store Operations, (con’t)

Instruction

Read Addr 1
Read Addr 2
Read Data 1
Read Data 2
Write Addr
Write Data

Register

File

ALU control

overflow
zero

RegWrite

ALU

MemWrite

Address

Read Data

Write Data

Data

Memory

Read Data

Sign
Extend

16

32

MemRead

C335 Computer Structures
Indiana University South Bend
Branch operations have to

- compare the operands read from the Register File during decode (rs and rt values) for equality (zero ALU output)
- compute the branch target address by adding the updated PC to the sign extended 16-bit signed offset field in the instruction
  - “base register” is the updated PC
  - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value and then shifted left 2 bits to turn it into a word address
Executing Branch Operations (con’t)

Branch target address

ALU control

(to branch control logic)
Jump operations have to replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits.
Creating a Single Datapath from the Parts

- Assemble the datapath elements, add control lines as needed, and design the control path
- Fetch, decode and execute each instructions in one clock cycle – single cycle design
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., why we have a separate Instruction Memory and Data Memory)
  - to share datapath elements between two different instruction classes will need **multiplexors** at the input of the shared elements with control lines to do the selection

- Cycle time is determined by length of the longest path
Multiplexor Insertion

Instruction Memory
- Read Address
- Read Instruction
- Write Address
- Write Data

ALU
- ALUSrc
- ALU control
- ovf
- zero

Register
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Data 2

Address
- Address
- Write Data

Data Memory
- Read Data

MemRead

RegWrite

Sign Extend
- 16
- 32

MemWrite
- MemtoReg

Add
- 4

PC

C335 Computer Structures
Adding the Branch Portion

Instruction Memory
- Read Address
- Read Instruction

Register File
- Read Addr 1
- Read Addr 2
- Write Addr
- Write Data

ALU
- ALUSrc
- ALU control
- ovf
- zero

Memory
- Read Data
- Write Data
- Address

ALU output
- Write Data
- MemWrite
- MemtoReg

PC
- 4

Add
- RegWrite

Shift left 2

Sign Extend
- 16
- 32

Data Memory
- Read Data

MemRead

C335 Computer Structures
Indiana University South Bend
Our Simple Control Structure

- We wait for everything to settle down
  - ALU might not produce “right answer” right away
  - Memory and RegFile reads are combinational (as are ALU, adders, muxes, shifter, signextender)
  - Use write signals along with the clock edge to determine when to write to the sequential elements (to the PC, to the Register File and to the Data Memory)

- The clock cycle time is determined by the logic delay through the longest path

We are ignoring some details like register setup and hold times
Register-Register Timing: One complete cycle

- Clk
- PC: Old Value ➔ New Value
- Rs, Rt, Rd, Op, Func: Old Value ➔ New Value
- ALUctr: Old Value ➔ New Value
- RegWr: Old Value ➔ New Value
- busA, B: Old Value ➔ New Value
- busW: Old Value ➔ New Value

Instruction Memory Access Time
Delay through Control Logic
Register File Access Time
ALU Delay

Register Write Occurs Here