Ch 5: Designing a Single Cycle Datapath

The Big Picture: Where are We Now?

• The Five Classic Components of a Computer

• Today’s Topic: Design a Single Cycle Processor

inst. set design (Ch 3) technology

machine design

Arithmetic (Ch 4)
The Big Picture: The Performance Perspective

• Performance of a machine is determined by:
  – Instruction count
  – Clock cycle time
  – Clock cycles per instruction

• Processor design (datapath and control) will determine:
  – Clock cycle time
  – Clock cycles per instruction

• Today:
  – Single cycle processor:
    • Advantage: One clock cycle per instruction
    • Disadvantage: long cycle time

How to Design a Processor: step-by-step

1. Analyze instruction set => datapath requirements
   – the meaning of each instruction is given by the *register transfers*
   – datapath must include storage element for ISA registers
     • possibly more
   – datapath must support each register transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effect the register transfer.
5. Assemble the control logic
The MIPS Instruction Formats

• All MIPS instructions are 32 bits long. The three instruction formats:

  - R-type
    
    | 31 | 26 | 21 | 16 | 11 | 6 | 0 |
    |----|----|----|----|----|----|----|
    | op | rs | rt | rd | shamt | funct |
    | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
  
  - I-type
    
    | 31 | 26 | 21 | 16 | 0 |
    |----|----|----|----|----|
    | op | rs | rt | immediate |
    | 6 bits | 5 bits | 5 bits | 16 bits |
  
  - J-type
    
    | 31 | 26 | 0 |
    |----|----|----|
    | op | target address |
    | 6 bits | 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

Step 1a: The MIPS-lite Subset

• ADD, SUB, AND, OR
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt

• LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16

• BRANCH:
  - beq rs, rt, imm16
Logical Register Transfers

• RTL gives the meaning of the instructions
• First step is to fetch the instruction from memory

\[
\begin{align*}
op &| rs &| rt &| rd &| shamt &| funct = MEM[PC] \\
op &| rs &| rt &| Imm16 & = MEM[PC]
\end{align*}
\]

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>SUB</td>
<td>R[rd] ← R[rs] − R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>OR</td>
<td>R[rt] ← R[rs]</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[ R[rs] + sign_ext(Imm16) ] ← R[rt]; PC ← PC + 4</td>
</tr>
</tbody>
</table>
| BEQ  | \begin{align*}
& \text{if ( R[rs] == R[rt] ) then } PC ← PC + \\
& \quad \text{sign_ext(Imm16)) || 00} \\
& \text{else } PC ← PC + 4
\end{align*}|

Step 1: Requirements of the Instruction Set

• Memory
  – instruction & data
• Registers (32 x 32)
  – read RS
  – read RT
  – Write RT or RD
• PC
• Extender
• Add and Sub register or extended immediate
• Add 4 or extended immediate to PC
Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements
  - Clocking methodology

Abstract/Simplified View of Datapath

- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)
Combinational Logic Elements (Basic Building Blocks)

- Adder
  - $A_{32}$
  - $B_{32}$
  - $\text{Sum}_{32}$
  - $\text{Carry}_{32}$

- MUX
  - Select
  - $A_{32}$
  - $B_{32}$
  - $Y_{32}$

- ALU
  - $A_{32}$
  - $B_{32}$
  - $\text{Result}_{32}$

State Elements: Review

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?

- falling edge
- rising edge
- cycle time
An unclocked state element

- The set-reset latch
  - output depends on present inputs and also on past inputs

Latches and Flip-flops

- Output is equal to the stored value inside the element
  (don’t need to ask for permission to look at the value)
- Change of state (value) is based on the clock
- Latches: whenever the inputs change, and the clock is asserted
- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

"logically true", could mean electrically low

A clocking methodology defines when signals can be read and written
— wouldn’t want to read a signal at the same time it was being written
D-latch

- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and its complement

D flip-flop

- Output changes only on the clock edge
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

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 Data In
Register File

- Built using D flip-flops

Note: we still use the clock to determine when to write
**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.”

**Storage Element: Idealized Memory**

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - 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.”
Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time

Step 3

- 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”
    - We don’t know if instruction is a Branch/Jump or one of the other instructions until we have fetched and interpreted the instruction from memory. So all instructions initially increment the PC
3b: R-format instructions: add, sub, and, or, slt

- \( R[rd] <- R[rs] \text{ op } R[rt] \)  
  - Example: add \( \text{ rd, rs, rt } \)
  - Read register 1, Read register 2, and Write register come from instruction’s rs, rt, and rd fields
  - ALU control and RegWrite: control logic after decoding the instruction

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
</tr>
<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>
<td></td>
</tr>
</tbody>
</table>
Datapath for R-format instructions

Register-Register Timing
3d: Load & Store Operations

- $R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]]$  
  Example: lw $rt$, $rs$, imm16
- Mem$[R[rs] + \text{SignExt}[imm16] \leftarrow R[rt]$  
  Example: sw $rt$, $rs$, imm16

\[
\begin{array}{c|c|c|c|c}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
\hline
6 \text{ bits} & 5 \text{ bits} & 16 \text{ bits} &
\end{array}
\]

- MemWrite
- Address
- Read data
- Write data
- Data memory

16

- Sign extend
- 32

a. Data memory unit  
b. Sign-extension unit

Datapath for lw & sw

- Instruction
- RegWrite
- Read register 1
- Read register 2
- Registers
- Write register
- Write data
- Read data 1
- Read data 2
- ALU operation
- ALU result
- Zero
- Sign extend
- 16
- 32
- Address
- Data memory
- Write data
- MemWrite
- MemRead

- Read data
- Read data
- Write data
3f: The Branch Instruction

<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>

- **beq rs, rt, imm16**
  - mem[PC] Fetch the instruction from memory
  - Equal <- 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 instruction
Using multiplexors to stitch together the datapath for memory access and R-format instructions

Putting it all together
Putting it all together

Adding the control unit
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.”

Critical Path (Load Operation) =
- PC’s Clk-to-Q +
- Instruction Memory’s Access Time +
- Register File’s Access Time +
- ALU to Perform a 32-bit Add +
- Data Memory Access Time +
- Setup Time for Register File Write +
- Clock Skew

Step 4: Given Datapath: RTL -> Control
Control

- Selecting the operations to perform (ALU, read/write, etc.)
  Design the ALU Control Unit
- Controlling the flow of data (multiplexer inputs)
  Design the Main Control Unit
- Information comes from the 32 bits of the instruction
- Example:

  add $8, $17, $18  Instruction Format:

  000000 10001 10010 01000 00000 100000
  op    rs    rt    rd   shamt  funct

- ALU's operation based on instruction type and function code

ALU Control

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

  35 2 1 100
  op rs rt 16 bit offset

- ALU control input

  000  AND
  001  OR
  010  add
  110  subtract
  111  set-on-less-than

- Why is the code for subtract 110 and not 011?
  (Recall design of ALU from Chapter 4. Bnegate input for adder set to 1 for subtraction)
ALU Control Design

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>Instruction operation</th>
<th>Funct field</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW 00</td>
<td></td>
<td>Load word</td>
<td>xxxxxx</td>
<td>Add</td>
<td>010</td>
</tr>
<tr>
<td>SW 00</td>
<td></td>
<td>Store word</td>
<td>xxxxxx</td>
<td>Add</td>
<td>010</td>
</tr>
<tr>
<td>BEQ 01</td>
<td></td>
<td>Branch eq</td>
<td>xxxxxx</td>
<td>Subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type 10</td>
<td></td>
<td>Add</td>
<td>100000</td>
<td>Add</td>
<td>010</td>
</tr>
<tr>
<td>R-type 10</td>
<td></td>
<td>Subtract</td>
<td>100010</td>
<td>Subtract</td>
<td>110</td>
</tr>
<tr>
<td>R-type 10</td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>And</td>
<td>000</td>
</tr>
<tr>
<td>R-type 10</td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>Or</td>
<td>001</td>
</tr>
<tr>
<td>R-type 10</td>
<td></td>
<td>Set on less than</td>
<td>101010</td>
<td>Set on less than</td>
<td>111</td>
</tr>
</tbody>
</table>

Control

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq
    - 10 = arithmetic
  - function code for arithmetic

- Describe it using a truth table (can turn into gates):

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp</th>
<th>Funct field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUOp1 ALUOp0</td>
<td>F5 F4 F3 F2 F1 F0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>X X X X X X</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>X 1</td>
<td>X X X X X X</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>X X 0 0 0 0</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>X X 0 1 0 1</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>X X 0 1 0 0</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>X X 0 1 0 1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>X X 1 0 1 0</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>
Design the main control unit

- Seven control signals
  - RegDst
  - RegWrite
  - ALUSrc
  - PCSrc
  - MemRead
  - MemWrite
  - MemtoReg

Control Signals

1. \( \text{RegDst} = 0 \) => Register destination number for the Write register comes from the \( rt \) field (bits 20-16)
   \( \text{RegDst} = 1 \) => Register destination number for the Write register comes from the \( rd \) field (bits 15-11)
2. \( \text{RegWrite} = 1 \) => The register on the Write register input is written with the data on the Write data input (at the next clock edge)
3. \( \text{ALUSrc} = 0 \) => The second ALU operand comes from Read data 2
   \( \text{ALUSrc} = 1 \) => The second ALU operand comes from the sign-extension unit
4. \( \text{PCSrc} = 0 \) => The PC is replaced with \( \text{PC} + 4 \)
   \( \text{PCSrc} = 1 \) => The PC is replaced with the branch target address
5. \( \text{MemtoReg} = 0 \) => The value fed to the register write data input comes from the ALU
   \( \text{MemtoReg} = 1 \) => The value fed to the register write data input comes from the data memory
6. \( \text{MemRead} = 1 \) => Read data memory
7. \( \text{MemWrite} = 1 \) => Write data memory
R-format instructions

| RegDst = 1 |
| RegWrite = 1 |
| ALUSrc = 0 |
| Branch = 0 |
| MemtoReg = 0 |
| MemRead = 0 |
| MemWrite = 0 |
| ALUOp = 10 |

Memory access instructions

<table>
<thead>
<tr>
<th>Load word</th>
<th>Store Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst = 0</td>
<td>RegDst = X</td>
</tr>
<tr>
<td>RegWrite = 1</td>
<td>RegWrite = 0</td>
</tr>
<tr>
<td>ALUSrc = 1</td>
<td>ALUSrc = 1</td>
</tr>
<tr>
<td>Branch = 0</td>
<td>Branch = 0</td>
</tr>
<tr>
<td>MemtoReg = 1</td>
<td>MemtoReg = X</td>
</tr>
<tr>
<td>MemRead = 1</td>
<td>MemRead = 0</td>
</tr>
<tr>
<td>MemWrite = 0</td>
<td>MemWrite = 1</td>
</tr>
<tr>
<td>ALUOp = 00</td>
<td>ALUOp = 00</td>
</tr>
</tbody>
</table>
Branch Equal

RegDst = X
RegWrite = 0
ALUSrc = 0
Branch = 1
MemtoReg = X
MemRead = 0
MemWrite = 0
ALUOp = 01
Step 5: Implementing Control

- Simple combinational logic (truth tables)

Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest 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.”

Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)
Summary

• 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

• MIPS makes it easier
  – Instructions same size
  – Source registers always in same place
  – Immediates same size, location
  – Operations always on registers/immediates

• Single cycle datapath => CPI=1, Clock Cycle Time => long