Design a MIPS Processor

• Instruction set overview of MIPS processors
• Single cycle MIPS processor
  – Datapath design
  – Controller design
• Multiple cycle MIPS Processor
  – Datapath design
  – Controller design
  • finite state machine; sequencer; microcode.
• Design a Multiple cycle MIPS Processor with Verilog at Behavioral/Structural Level (Project 5)

MIPS ISA as an Example

• Instruction categories:
  – Load/Store
  – Computational
  – Jump and Branch
  – Floating Point
  – Memory Management
  – Special

3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>R type</th>
<th>OP</th>
<th>SrS</th>
<th>Srt</th>
<th>Srd</th>
<th>sa</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type</td>
<td>OP</td>
<td>SrS</td>
<td>Srt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>Jump</td>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
</tr>
</tbody>
</table>

MIPS Arithmetic Instructions

• MIPS assembly language arithmetic statement
  add$r10,$r11,$r12
  sub$r10,$r11,$r12

• Each arithmetic instruction performs only one operation
• Each arithmetic instruction fits in 32 bits and specifies exactly three operands
  destination ← source1 op source2
• Those operands are all contained in the datapath’s register file ($r10,$r11,$r12) – indicated by $
• Operand order is fixed (destination first)
MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory:
  
  ```
  lw $t0, 4($s3)  # load word from memory
  sw $t0, 8($s3)  # store word to memory
  ```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5-bit address.

- The memory address – a 32-bit address – is formed by adding the contents of the base address register to the offset value:
  
  - A 16-bit field meaning access is limited to memory locations within a region of \( \pm 2^{13} \) or 8,192 words (\( \pm 2^{15} \) or 32,768 bytes) of the address in the base register.
  
  - Note that the offset can be positive or negative.

MIPS Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>$t3, 500($t4) Store word</td>
</tr>
<tr>
<td>sh</td>
<td>$t3, 502($t2) Store half</td>
</tr>
<tr>
<td>sb</td>
<td>$t2, 41($t3) Store byte</td>
</tr>
<tr>
<td>lw</td>
<td>$t1, 30($t2) Load word</td>
</tr>
<tr>
<td>lh</td>
<td>$t1, 40($t3) Load halfword</td>
</tr>
<tr>
<td>lh</td>
<td>$t1, 4($t3) Load halfword unsigned</td>
</tr>
<tr>
<td>lb</td>
<td>$t1, 40($t3) Load byte</td>
</tr>
<tr>
<td>lbu</td>
<td>$t1, 40($t3) Load byte unsigned</td>
</tr>
<tr>
<td>lui</td>
<td>$t1, 40 Load Upper Immediate (16 bits shifted left by 16)</td>
</tr>
</tbody>
</table>

MIPS Control Flow Instructions

- MIPS conditional branch instructions:
  
  ```
  bne $s0, $s1, Lbl  # go to Lbl if $s0 != $s1
  beq $s0, $s1, Lbl  # go to Lbl if $s0 = $s1
  ```

  - Ex: if (i==j) h = i + j;
    
    ```
    bne $s0, $s1, Lbl1
    add $s3, $s0, $s1
    Lbl1: ...
    ```

- Instruction Format (I format):
  
  ```
  | op | rs | rt | 16 bit offset |
  ```

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset:
  
  - which register? Instruction Address Register (the PC)
  
  - Its use is automatically implied by instruction
  
  - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction.

  - Limits the branch distance to \( \pm 2^{16} \) to \( +2^{16}-1 \) instructions from the (instruction after the) branch instruction.

  ![Branch Destination Diagram](image)
Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:
  
  ```
  j label //go to label
  ```

- Instruction Format (J Format):
  
  ```
  op 26-bit address
  ```

  from the low order 26 bits of the jump instruction

MIPS Immediate Instructions

- Small constants are used often in typical code
  
  ```
  addi $sp, $sp, 4  //sp = sp + 4
  slti $t0, $s2, 15  //t0 = 1 if s2<15
  ```

- Machine format (I format):

  ```
  op rs rt 16 bit immediate I format
  ```

  The constant is kept inside the instruction itself!
  
  - Immediate format limits values to the range $2^{16}$–1 to -$2^{16}$

  more MIPS Immediate Instructions:
  
  ```
  addi $r29, $r29, 4
  slti $r8, $r18, 10
  andi $r29, $r29, 6
  ori $r29, $r29, 4
  addi $r10,$r111,10
  ```

R-Format Example

- MIPS Instruction:
  
  ```
  add   $8,$9,$10
  ```

  - opcode = 0 (look up in table)
  - funct = 32 (look up in table)
  - rs = 9 (first operand)
  - rt = 10 (second operand)
  - rd = 8 (destination)
  - shamt = 0 (not a shift)

  binary representation:

  
  | 000000 | 01001 | 01010 | 01000 | 00000 | 100000 |

I-Format Example 1

- MIPS Instruction:
  
  ```
  addi $21,$22,-50
  ```

  - opcode = 8 (look up in table)
  - rs = 22 (register containing operand)
  - rt = 21 (target register)
  - immediate = -50 (by default, this is decimal)

  decimal representation:

  | 8  | 22 | 21 | -50 |

  binary representation:

  | 001000 | 10110 | 10101 | 111111110011110 |
I-Format Example 2

- MIPS Instruction:
  - `lw $8,1200($9)`
  - opcode = 35 (look up in table)
  - rs = 9 (base register)
  - rt = 8 (destination register)
  - immediate = 1200 (offset)

Decimal representation:

<table>
<thead>
<tr>
<th>decimal representation:</th>
<th>35</th>
<th>9</th>
<th>8</th>
<th>1200</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary representation:</td>
<td>100011</td>
<td>01001</td>
<td>01000</td>
<td>0000010010110000</td>
</tr>
</tbody>
</table>

Our Example: A MIPS Subset

- R-Type:
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt
  - slt rd, rs, rt

- Load/Store:
  - lw rt, rs, imm16
  - sw rt, rs, imm16

- Imm operand:
  - addi rt, rs, imm16

- Branch:
  - beq rs, rt, imm16

- Jump:
  - j target

Register Transfers

- RTL gives the meaning of the instructions
- All start by fetching the instruction, read registers, then use ALU. Memory access, write results.

MEM[PC] = op | rs | rt | rd | imm16 | funct
or = op | rs | rt | Imm26

Inst Register transfers

ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4
SUB R[rd] <- R[rs] - R[rt]; PC <- PC + 4
LOAD R[rt] <- MEM[R[rs] + sign_ext(Imm16)]; PC <- PC + 4
STORE MEM[R[rs] + sign_ext(Imm16)] <- R[rt]; PC <- PC + 4
ADDI R[rt] <- R[rs] + sign_ext(Imm16); PC <- PC + 4
BEQ if (R[rs] == R[rt]) then PC <- PC + 4 + sign_ext(Imm16) || 00  
else PC <- PC + 4
Requirements of the Datapath

After checking the register transfers, we can see that datapath needs the followings:

• Memory
  – store instructions and data
• Registers (32 x 32)
  – read RS
  – read RT
  – Write RT or RD
• PC
• Extender for zero- or sign-extension
• Add and sub register or extended immediate (ALU)
• Add 4 or extended immediate to PC

Datapath Components

Storage elements:
- 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

- 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 and puts the content on busA (data)
  – RB selects the register and puts the content on busB (data)
  – RW selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
- Can have other control signals in your design
  – such as read, write, etc
Clocking Methodologies
- The clocking methodology defines when signals can be read and when they are written.
  - An edge-triggered methodology
- Typical execution
  - read contents of state elements
  - send values through 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 to hold the address of the next instruction
  - PC is updated every cycle, so it does not need an explicit write control signal
  - Instruction Memory is read every 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

Executing R Format Operations
- R format operations (add, sub, slt, and, or)
  - perform the (op and funct) operation on values in rs and rt
  - store the result back into the Register File (into location rd)
  - The Register File is not written every cycle (e.g. sw), so we need an explicit write control signal for the Register File
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File

Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (zero ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instruction

Executing Jump Operations

- Jump operation involves
  - 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 segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path
Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)

R-type:

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>funct</td>
<td>shamt</td>
<td></td>
</tr>
</tbody>
</table>

I-Type:

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>20</th>
<th>15</th>
<th>10</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>address offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

J-type:

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>target address</td>
<td></td>
</tr>
</tbody>
</table>

Observations

- op field always in bits 31-26
- addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
- addr of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
- offset for beq, lw, and sw always in bits 15-0
Adding the Jump Operation

Read Address Instr[31-0]
Instruction Memory
Add PC
4
Write Data
Read Addr 1
Read Addr 2
Write Addr
Register File
Read Data 1
Read Data 2
ALU
ovf
zero
RegWrite
Data Memory
Address
Write Data
Read Data
MemWrite
MemRead
Sign Extend
16 32
MemtoReg
ALUSrc
Shift left 2
Add
PCSrc
Jump
Branch
Multicycle Datapath Approach
• Let an instruction take more than 1 clock cycle to complete
  – Break up instructions into steps where each step takes a cycle while trying to
    • balance the amount of work to be done in each step
    • restrict each cycle to use only one major functional unit
  – Not every instruction takes the same number of clock cycles
• In addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on different clock cycles, as a result
  – only need one memory – but only one memory access per cycle
  – need only one ALU/adder – but only one ALU operation per cycle

Single Cycle Disadvantages & Advantages
• Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  – especially problematic for more complex instructions like floating point multiply

Cycle 1
lw
sw
Cycle 2
Waste
• May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle
  but
• Is simple and easy to understand

Multicycle Datapath Approach
• At the end of a cycle
  – Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)
  – Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

IR – Instruction Register
MDR – Memory Data Register
A, B – regfile read data registers
ALUout – ALU output register

Waste Cycle 1
Cycle 2
lw
sw
The Multicycle Datapath with Control Signals

Multicycle Control Unit

- Multicycle datapath control signals are not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but not what instruction cycle is to be done next
- Must use a finite state machine (FSM) for control
  - a set of states (current state stored in State Register)
  - next state function (determined by current state and the input)
  - output function (determined by current state and the input)

The Five Steps of the Load Instruction

- Ifetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES

Multicycle Advantages & Disadvantages

- Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction
- Multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles
  - but
- Requires additional internal state registers, more muxes, and more complicated (FSM) control