CHAPTER 5
THE PROCESSOR: DATAPATH and CONTROL

The performance of a computer is determined by:

a) The instruction count ----determined by the compiler used and instruction set
b) Clock cycle time (clock rate or frequency) determined by the implementation
c) Clock cycles per instruction (CPI) of the processor

• We will design an implementation that includes the core of the MIPS instruction set including:
  a) The memory-reference instructions: lw and sw
  b) The arithmetic-logical instructions: add, sub, and, or, slt
  c) The branch if equal (beq) and jump (j) instructions

  Focus will be to illustrate the key principles and methods used in creating a datapath and designing the control unit.

• To implement an instruction: (common steps) are:

  1. Fetch the instruction from memory by using the PC.
  2. Looking at the fetched instruction fields, read one or two registers (e.g. for load: read one register, for others: read two registers) [ Decode the inst.]
  3. After reading the registers, use the ALU:
     - memory-ref. instructions use the ALU for effective address calculation
     - arithm-logical instructions use the ALU for op-code execution
     - branch instructions use the ALU for comparison

• After using the ALU, actions required depend on the instruction type:
  - memory-ref. inst. : need to access the memory (to load or store a word)
  - arithm-logical inst. : must write the data from ALU back into a register.
  - branch inst. : may need to modify the next instruction address based on the comparison result.
An abstract view of the MIPS implementation showing the major functional units, and connections between them is as follows:

Logic conventions and clocking:

assert : logically high
combinational blocks : have no memory, their outputs depend only on current inputs.
sequential blocks : or state elements have memory

Example: D-FF

We can construct memory elements (for data and instruction memory) and registers using sequential blocks. Their outputs depend both on the current inputs, and their internal state (previous state).

Clocking:
Distinguish between “reading” and “writing” of signals. If a signal is written at the same time it is read, the value read could be the old value, the new value, or a mixture of both!
⇒ Assume edge-triggered clocking where the updates are done only on a clock edge:
Ex.1:

```
<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>State element 2</td>
</tr>
</tbody>
</table>
```

State element 2 looks at its inputs and changes state here! Use inputs, produce outputs one clock cycle.

Ex.2: If one clock cycle is not enough for the signals to propagate from state-element 1 through combinational block ⇒ a Write signal is needed

```
<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Write</td>
</tr>
</tbody>
</table>
```

The state elements are updated on clock edges, but only if the “Write” signal is also asserted.

It is also possible to read the contents of a register, use some combinational logic to determine the next contents of that register, and then write the register in the same clock cycle, at the clock edge:

```
| Register | Combinational Block |
```

**Bus**: a signal connection wider than 1 bit

Ex.:  

```
Reg1 ---- 32 ---- Reg1
```

**Counter**: a register with special connections between flip-flops so that its latched value increments on every active clock cycle.

**Multiplexer**: It is a combinational circuit to select one of many inputs.
**MIPS Subset Implementation:**

- Simple implementation that uses a single long clock cycle for every instruction:

  ![Simple Implementation Diagram]

  This is not very realistic, and would waste time for instructions that require shorter time. However, it produces a much simpler control circuitry.

**BUILDING SINGLE CLOCK DATAPATH:**

Elements needed:
1. A memory unit (Instruction memory) to hold instructions [sequential]
2. A Program Counter (PC) to keep the address of the next instruction [sequential]
3. An adder to increment the PC to the address of the next instruction [combinational]

**R-TYPE INSTRUCTIONS**

At the starting point of the implementation, we will target only *R-type* arithmetic-logic instructions, *add, sub, and, or, slt*.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>(6 bits) b&lt;sub&gt;31..26&lt;/sub&gt;</th>
<th>(5 bits) b&lt;sub&gt;25..21&lt;/sub&gt;</th>
<th>(5 bits) b&lt;sub&gt;20..16&lt;/sub&gt;</th>
<th>(5 bits) b&lt;sub&gt;15..11&lt;/sub&gt;</th>
<th>(5 bits) b&lt;sub&gt;10..6&lt;/sub&gt;</th>
<th>(6 bits) b&lt;sub&gt;5..0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>sub $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>and $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>or $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>slt $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>42</td>
</tr>
</tbody>
</table>

The algorithm of the implementation is as follows:

1. To execute an instruction, we start by fetching it from the instruction memory, and we then increment the PC so that it points to the next instruction (+4 bytes).

   ⇒ The datapath for this step is:

   ![Increment PC by four Diagram]
2- Access to the instruction in the instruction memory pointed by PC and decode the fields of the instruction to determine
   i- the numbers of the source registers ($rs$, and $rt$).
   ii- the number of the destination register $rd$.
   iii- the operation specified in the instruction, from $opc$ and $fn$.

3- Get the contents of the registers, $rs$ and $rt$, from the register file and transfer them to ALU.

4- Perform the specified operation in ALU and write the result of ALU operation to the destination register which is pointed by $rd$. 

To read: supply reg. #, (to one read reg. input port), get contents of that reg. at output.
To write: supply reg. # to the write reg. input port, supply data to be written into that reg. to the write data input port. Then, assert the “Regwrite” control signal.

- This data-path organization can perform all *R-type* instructions. The implementation of the controller part of the ASM chart related to this data-path does not need a state element.

ASM chart of the only-R-type instruction computer with single-clock cycle implementation. A single-block ASM chart corresponds to a combinatory circuit.

**IMMEDIATE MEMORY-REGISTER TRANSFER INSTRUCTIONS**

Now, consider the MIPS *load* and *store* instructions

<table>
<thead>
<tr>
<th>I-format Instructions</th>
<th>Fields</th>
<th>(6-bit)</th>
<th>(5-bit)</th>
<th>(5-bit)</th>
<th>(16-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I &amp; J</td>
<td>b31..b26</td>
<td>b25..b21</td>
<td>b20..b16</td>
<td>b15..b0</td>
</tr>
<tr>
<td></td>
<td>opc</td>
<td>rs</td>
<td>rt</td>
<td>imm</td>
<td>value or address</td>
</tr>
<tr>
<td><em>lw</em> $1, offset($2)</td>
<td>35</td>
<td>2</td>
<td>1</td>
<td>offset</td>
<td></td>
</tr>
<tr>
<td><em>sw</em> $1, offset($2)</td>
<td>43</td>
<td>2</td>
<td>1</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

- Note that *lw* instruction specifies a destination and a source register, while both of the registers specified by *sw* are source registers. Thus, we shall be able to transfer *rt* to *rreg2* for *sw*, or *wreg* for *lw*, depending on the opcode of the instruction. We can accomplish it using a multiplexer at the *wreg* input of the register-file.
- So, the register file and ALU we used for the R-type instructions are also required for these instructions. An extra unit will be needed to sign-extend the 16-bit offset field to a 32-bit signed value (in 2’s complement form), and a data memory unit.
- To implement the memory-transfer instructions in single-clock-cycle, we shall use an isolated data-memory.

1) For both *lw* and *sw*, ALU shall calculate the data-memory address.
2) *lw* requires datapath to transfer the memory contents to the *write-data* input of the register-file. Control unit access the contents by asserting *MemRead* signal to the data-memory on the opcode of *lw* instruction.

3) *sw* requires a path to write $rt$ (the contents of $rt$) to the memory-location with the calculated address. Control unit asserts *MemWrite* signal to the data-memory on the opcode of *sw* instruction.
How to combine datapaths for the Memory Instructions and R-type instructions?

Answer:

To combine the datapaths of the individual instruction classes we considered, into a single datapath. First, consider an implementation that uses a single clock cycle for every instruction. This means: no datapath section can be used more than once per instruction. So, any element needed more than once must be duplicated.

- We have to insert multiplexers into the circuit to select one of the lines connected to the \( wreg \), \( wdata \), and \( ALU-B \) inputs depending on the instruction.

ASM-chart to implement LW, SW and R-type instruction format.
**Branch Instruction**

Now we will expand the datapath diagram to implement branch and jump instructions. "branch if equal", \( beq \) instruction is \( I\)-type,

\[
\text{\textbf{Fields}} \quad b_{31..26} \quad \text{(6-bit)} \quad b_{25..21} \quad \text{(5-bit)} \quad b_{20..16} \quad \text{(5-bit)} \quad b_{15..0} \quad \text{(16-bit)}
\]

\[
\text{Instruction} \quad \text{op} \quad \text{rs} \quad \text{rt} \quad \text{imm} \quad \text{address or value}
\]

```
beq $1,$2,address
```

“\( beq \)” changes the contents of \( PC \) register if the contents of the specified registers \( rs \) and \( rt \) are exactly the same. \( PC \) register contains byte address, but the immediate value contains \( \text{PC-relative-word address} \), and thus it must be converted to \( \text{PC-relative-byte-address} (=\text{imm} \times 4) \) before adding on the contents of \( PC \).

\[
\begin{align*}
\text{if} \ ( \text{\$rs} = \text{\$rt}) \text{ then } \text{PC} & \leftarrow \ (\text{PC} + 4) + \text{imm} \times 4 \\
\text{else} \ \text{PC} & \leftarrow \ \text{PC} + 4
\end{align*}
\]

Note that \( \text{imm} \times 4 \) is 2-bit left-shifted form of \( \text{imm} \). To have a 32-bit result, the 16-bit signed binary number \( \text{imm} \) it must be sign-extended before the shift-left.

![Circuit level realization of the sign-extend and shift-left-2 operations.](image1)

![New datapath for BEQ instruction requires an additional ALU.](image2)
Instructions performed by the datapath diagram.

<table>
<thead>
<tr>
<th>fields:</th>
<th>(b_{31:26})</th>
<th>(b_{25..21})</th>
<th>(b_{20:16})</th>
<th>(b_{15..11})</th>
<th>(b_{10..6})</th>
<th>(b_{5..0})</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>opc</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>sh</td>
<td>fn</td>
</tr>
<tr>
<td>add $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>sub $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>and $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>or $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>slt $1,$2,$3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>l-format</td>
<td>opc</td>
<td>rs</td>
<td>rt</td>
<td>imm: (value or address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq $1,$2,400</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $1,100($2)</td>
<td>35</td>
<td>2</td>
<td>1</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $1,100($2)</td>
<td>43</td>
<td>2</td>
<td>1</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The overall datapath diagram for RT, lw, sw, beq instructions.

ASM-chart of the datapath diagram to perform R-type, lw, sw, beq instructions.

reset

opc =00 0000:

opc =00 0100:

opc =01 0011:

opc =01 0111:

opc =01 1011:

all other opcodes perform NOP: no-operation
ALU Control
ALU-control unit is a combinational block designed to simplify the functions of the main control unit. Main control unit uses the opc-field, and decides on the ALUOp input of the ALU-control unit.

- For the LW and SW instructions ALU should compute the effective memory address by addition ( $rs + sign-extended-imm$).
- For arithmetic-logical instructions (RF) ALU should operate according to the 6-bit fn field in the instruction.
- For the BEQ instruction, ALU should perform a subtraction ($rs$–$rt$).

So, design a small control unit which generates the 3-bit ALU control input signals. We shall use a 2-bit ALUOp control input, which will be used to show whether the operation to be performed is an add, for load/store, (00), subtract for beq (01), or the operation given in the function field (10).

(ALUOp bits are actually generated by the main control unit: to be discussed later)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Type</th>
<th>A1,A0</th>
<th>F5 .. F0</th>
<th>Op2 .. Op0</th>
<th>ALU-function</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>I</td>
<td>0 0</td>
<td>xx xxxx</td>
<td>0 1 0</td>
<td>add: (A+B)</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>0 0</td>
<td>xx xxxx</td>
<td>0 1 0</td>
<td>add: (A+B)</td>
</tr>
<tr>
<td>beq</td>
<td>I</td>
<td>0 1</td>
<td>xx xxxx</td>
<td>1 1 0</td>
<td>subtract: (A–B)</td>
</tr>
<tr>
<td>add</td>
<td>R</td>
<td>1 0</td>
<td>10 0000</td>
<td>0 1 0</td>
<td>add: (A+B)</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>1 0</td>
<td>10 0010</td>
<td>1 1 0</td>
<td>sub: (A–B)</td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>1 0</td>
<td>10 0100</td>
<td>0 0 0</td>
<td>and: (A and B)</td>
</tr>
<tr>
<td>or</td>
<td>R</td>
<td>1 0</td>
<td>10 0101</td>
<td>0 0 1</td>
<td>or: (A or B)</td>
</tr>
<tr>
<td>slt</td>
<td>R</td>
<td>1 0</td>
<td>10 1010</td>
<td>1 1 1</td>
<td>slt: sign(A–B)</td>
</tr>
</tbody>
</table>

In the truth table, the leftmost-2-bits of funct is always the same value, and has no significance for the Boolean functions. When AOp1=0 (i.e., AOp=00 or =01) the ALU-control signal does not depend on funct-field at all. We can split the truth table into two independent parts using only 4-bits of fn-field:

While A1=1, in the second table A0 is always 0 and is insignificant (marked as don't care). Therefore Op depends on F3 .. F0 only.
While $A1=0$, from the first table

\[ \text{Op2 = A0} \quad ; \quad \text{Op1 = A1}' \]

When we combine these functions using OR operator we obtain

\[
\begin{array}{c|c|c}
\text{Op2} &=& A0 + F1 A1 \\
\text{Op1} &=& A1' + F2' A1 \\
\text{Op0} &=& (F3 + F0) A1
\end{array}
\]

Op1 is further simplified to

\[
\]

Let's write Boolean expressions in terms of fn and ALUOp:

\[ \text{Op0} = (fn3 + fn0) \text{ ALUOp1}; \]
\[ \text{Op1} = (ALUOp1 \ fn2)'; \]
\[ \text{Op2} = \text{ALUOp0} + fn1 \text{ ALUOp1}. \]