Multi-Cycle Design Principles

- Break up execution of each instruction into steps.
- The number of steps and the tasks in each step are instruction dependent.
- Each step takes one clock cycle.
- Balance the amount of work to be done in each clock cycle.
- Restrict each cycle to use only one major functional unit in the data path, or if more than one major functional unit used they should be used in parallel.
- Major units are memory, register file and ALU, since we assume that they introduce the most significant delays during execution of instructions.
- We assume all other delays in the wiring is negligible.
Multi-Cycle Design Principles (cont.)

- During execution of any instruction, we may be reusing functional units, but in different steps (clock cycles), e.g.
  - Single memory can be used for instruction and data,
  - ALU will be used to compute not only tasks it performed in the single-cycle design (e.g. lw & sw addresses and R-type instruction calculations), but it will be used to increment PC (by 4) and to calculate branch target address.

- Control signals will not be determined solely by the instruction in execution (i.e. its op-code and/or function code) but also by the particular clock cycle the instruction is being executed in.

- At the end of each cycle during instruction execution store intermediate values for use in later cycles.

- For that purpose, introduce additional “internal” registers.

Elaboration on Work Balance in Each Step

- During any given step it is not allowed to have a serial combination of usage of the major functional units; for example:
  - It is not allowed that in one step contents of registers are read from the register file and then those contents are used as operands for ALU in the same step, or
  - It is not allowed that in one step ALU performs a function on some operands and its result is used as an address for memory read or write in the same step.

- This principle is introduced to avoid that any step requires too much time, implying that clock cycles have to be of that unnecessary length.

- Notice that two of the major functional units are allowed to be used in parallel, e.g. reading contents from a register file and the ALU performing a function on unrelated data at the same time.
**Five Steps In Instruction Execution**

- Major steps in execution of an instruction are:
  - Instruction Fetch
  - Instruction Decode and Register Fetch
  - Execution, Memory Address Computation, or Branch Completion
  - Memory Access or R-type instruction completion
  - Write-back step
- Not every instruction will have all those steps
- Our instructions will take 3-5 steps, i.e. 3-5 clock cycles.
- The first two steps are common to all instructions.

---

**Multi-Cycle Datapath High Level View**

- The use of shared functional units requires new temporary registers that hold data between clock cycles of the same instruction.
- The additional registers are:
  - Instruction register (IR),
  - Memory data register (MDR),
  - A and B registers,
  - ALUout register.
Multi-Cycle Datapath Detailed View

Figure 5.27
with additions in red

Multi-Cycle Datapath and Control

Figure 5.28
Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register, i.e. \( IR \leftarrow \text{Memory}[\text{PC}] \); \( \text{IRd}=0, \text{MemRead, IRWrite} \)

- Increment the PC by 4 and put the result back in the PC, i.e. \( PC \leftarrow [\text{PC}] + 4 \); \( \text{ALUSrcA}=0, \text{ALUSrcB}=01, \text{ALUOp}=00, \text{PCSource}=0, \text{PCWrite} \)

- Here are rules for signals that are omitted:
  - If signal for mux is not stated, it is don’t care
  - If ALU signals are not stated, they are don’t care
  - If MemRead, MemWrite, RegWrite, IRWrite, PCWrite or PCWriteCond is not stated, it is unasserted, i.e. logical 0.

Step 2: Instruction Decode & Register Fetch

- We aren’t setting any control lines based on the instruction type, since we are busy "decoding" it in our control logic.

- Read registers rs and rt in case we need them:
  A \( \leftarrow \text{Reg}[\text{IR}[25-21]] \);
  B \( \leftarrow \text{Reg}[\text{IR}[20-16]] \);
  Done automatically

- Compute the branch address in case the instruction is a branch:
  \( \text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend(}\text{IR}[15-0]) << 2) \);
  \( \text{ALUSrcA}=0, \text{ALUSrcB}=11, \text{ALUOp}=00 \)
Step 3: Execute, Mem Addr, Branch

- ALU is performing one of three functions, based on instruction type
- Memory Reference (lw or sw):
  ALUOut ← A + sign-extend(IR[15-0]);
  ALUSrcA=1, ALUSrcB=10, ALUop=00
- R-type:
  ALUOut ← A op B;
  ALUSrcA=1, ALUSrcB=00, ALU0p=10
- Branch on Equal:
  if (A==B) PC ← ALUOut;
  ALUSrcA=1, ALUSrcB=00, ALU0p=01
  PCSource=01, PCWriteCond

Note: beq instruction is done, thus this instruction requires 3 clock cycles to execute.

Steps 4 and 5: Instruction Dependent

- Step 4: R-type and Memory Access
  - Loads and stores access memory
    MDR ← Memory[ALUOut] (load);  IorD=1, MemRead
    or
    Memory[ALUOut] ← B (store);  IorD=1, MemWrite
  - R-type instructions finish
    Reg[IR[15-11]] ← ALUOut;  RegDst=1, MemToReg=0, RegWrite

  Register write actually takes place at the end of the cycle on the falling edge
  - Store and R-type instructions are done in 4 clock cycles

- Step 5: Write back (load only)
  - Reg[IR[20-16]] ← MDR  RegDst=0, MemToReg=1, RegWrite
Summary of Instruction Executions

Figure 5.30

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR ← Memory[PC]</td>
<td></td>
<td></td>
<td>PC ← PC + 4</td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>A ← Reg [IR[25-21]] B ← Reg [IR[20-16]]</td>
<td>ALUOut ← PC + (sign-extend [IR[15-0]] &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut ← A op B</td>
<td>ALUOut ← A + sign-extend [IR[15-0]]</td>
<td>If (A == B) then PC ← ALUOut</td>
<td>PC ← PC [31-28] if (IR[25-0] &lt;&lt; 2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] ← ALUOut</td>
<td>Load: MDR ← Memory[ALUOut] or Store: Memory [ALUOut] ← B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20-16]] ← MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Jump instruction added: PCSource=10, PCWrite

Implementing Control

- Values of control signals are dependent on:
  - what instruction is being executed and
  - which step (i.e. clock cycle) is being performed.

- Use the information we’ve accumulated to specify a finite state machine – FSM:
  - specify the finite state machine graphically, or
  - use microprogramming.

- Then, an implementation can be derived from specification.
Finite State Machines

- A current state is kept in the Current state register.
- Next state function and Output function are determined by Current state and Inputs.
- In our case, Output function is based only on Current state.
MIPS Interrupt Processing

We are implementing processing of only two exceptions:

– illegal op-code and
– integer overflow.

When any of the exceptions occurs, MIPS processor processes the exception (as any other interrupt) in the following 3 steps:

**Step 1:** EPC register gets a value equal to address of a faulty instruction.

**Step 2.**: \( \text{PC} \leftarrow 80000180_{16} \)

Cause register \( \leftarrow \) a code of the exception

– illegal op-code = 10
– integer overflow = 12

**Step 3.** Processor is now running in Kernel mode.

Note: we are not implementing step 3.
Multi-Cycle Datapath for Exception Handling

![Diagram of Multi-Cycle Datapath for Exception Handling]

Figure 5.39
with corrections in red

FSM Graph with Exception Handling

![Diagram of FSM Graph with Exception Handling]

Figure 5.40
with additions in red