Drawbacks of Single-Cycle Implementation

All instructions must complete in 1 cycle (CPI = 1)
- different instructions do different amounts of work, for example:
  - `add` uses instruction memory, ALU, register file twice
  - `lw` also uses these + data memory
  - clock cycle set to the longest instruction

Hardware units can only be used once in the cycle
- some must be replicated (ALU, memory)
- increased hardware costs

Alternative to Single-Cycle Implementation

Multicycle implementation
- Each instruction executes in multiple shorter cycles
- Each instruction takes as many cycles as it needs to get its work done
- Length of a cycle is determined by the delay of individual functional units
- Fewer resources if some can be reused in different cycles
Multiple-cycle Implementation

Break up the execution cycle into steps:
- want each step to contain work that takes about the same amount of time
- instructions only use the steps they need

1. instruction fetch
2. instruction decode & source register(s) read
3. ALU execution
4. memory access (read/write) or ALU completion (write the result register)
5. write back register for a load

Hardware Changes

Add some temporary registers (not visible in the ISA) since some information that is calculated in one cycle is needed in subsequent cycles
- instruction register (IR)
- memory data register (MDR)
- ALU source registers, A and B
- ALUOut

Data that is calculated in 1 instruction & needed by subsequent instructions is stored in ISA-visible state (PC, registers, memory)

Larger or more MUXes
- MUX to memory address
- MUX to ALU source 1
- larger MUX to ALU source 2
**Instruction Fetch**

Actions:

- IR <-- Memory[PC]
- PC <-- PC + 4

Implementation registers:

- instruction register: information will be needed in subsequent cycles

Hardware that is shared in different cycles

- memory (data memory later)
- ALU to increment the PC

---

**Instruction Decode & Source Register(s) Read**

Actions:

- A <-- Register[IR[25:21]] (read rs)
- B <-- Register[IR[20:16]] (read rt)
- ALUOut <-- PC + sign-extend IR[15:0] << 2
  (performed early in case this instruction is a branch)

Implementation registers:

- register A
- register B
  both needed as operation source operands in the next cycle
- ALUOut for the target address

Hardware that is shared in different cycles

- ALU to calculate branch target
**ALU Execution**

Actions:
- if R-type instruction
  \[\text{ALUOut} \leftarrow A \text{ op } B\]
- if data transfer instruction
  \[\text{ALUOut} \leftarrow A + \text{sign-extend (IR}[15:0])\]
- if branch instruction (\& successful)
  \[\text{if } (A == B) \text{ PC} \leftarrow \text{ALUOut}\]
  (this is the value of ALUOut computed on the last cycle)

Implementation registers:
- ALUOut passes the target address from the last step

Hardware that is shared in different cycles
- ALU

---

**Memory Access or Write an ALU Result**

Actions:
- if load instruction
  \[\text{memory data register (MDR)} \leftarrow \text{Memory[ALUOut]}\]
- if store instruction
  \[\text{Memory[ALUOut]} \leftarrow B\]
- if R-type instruction
  \[\text{Register}[\text{IR}[15:11]] \leftarrow \text{ALUOut}\]

Implementation registers:
- MDR
- ALUOut

Hardware that is shared in different cycles
- ALU
- Memory
Load Completion

Actions:

Register[IR[20:16]] <-- Memory data register (MDR)

Implementation registers:

• MDR

Performance Example

Multiple-cycle implementation has better performance because each instruction takes only as many cycles as it needs

Example:

• cycles per instruction
  loads: 5, stores: 4, R-type: 4, branches: 3
  percentage in total instructions
  loads: 22%, stores: 11%, R-type: 50%, branches: 17%
• both implementations have the same number of instructions
• CPI_{single} = 5
• CPI_{multi} = 5*.22 + 4*.11 + 4*.50 + 3*.17 = 4.05
• speedup = 5/4.05 = 1.2
Multiple-cycle Implementation: Control

Control is more complex than in a single-cycle implementation
• need to define control signals for each step
• need to know which step we’re on

Two implementations for the control unit
• **hardwired control**
  • specified as a finite state machine (FSM)
• **microprogramming**
  • expressed as a “micro” programming language

Both specifications can be synthesized into hardware

---

Instruction Fetch

Set the MUX so that the PC is the memory address:
\[ \text{IorD} = 0 \]
Set **MemRead** signal
Set **IRWrite** signal
Set the MUX for ALU source 1 to be from the PC:
\[ \text{ALUSrcA} = 0 \]
Set the MUX for ALU source 2 to be from the constant 4:
\[ \text{ALUSrcB} = 01 \]
Set ALUcontrol to “+”:
\[ \text{ALUOp} = 00 \]
Set the MUX for input to the PC to be from the ALU:
\[ \text{PCSource} = 00 \]
Set **PCWrite**

Why do we need a signal to write the IR?

The ALU result is also stored in ALUout: why does this not matter?

The PC can be incremented & the memory accessed for an instruction during the same cycle: why can this be done?
Instruction Decode & Read Source Register(s)

Set the MUX for ALU source 1 to be from the PC:
\[ ALUSrcA = 0 \]

Set the MUX for ALU source 2 to be from the sign-extended, shifted immediate:
\[ ALUSrcB = 11 \]

Set ALUcontrol to "+":
\[ ALUOp = 00 \]

When are temporary registers A and B written?

What if this turns out not to be a branch instruction?

Execute

Which control signals are generated depends on the opcode

- data transfer
  - Set the MUX for ALU source 1 to be from register A:
    \[ ALUSrcA = 1 \]
  - Set the MUX for ALU source 2 to be from the sign-extended immediate:
    \[ ALUSrcB = 10 \]
  - Set ALUcontrol to "+":
    \[ ALUOp = 00 \]

- R-type
  - Set the MUX for ALU source 1 to be from register A:
    \[ ALUSrcA = 1 \]
  - Set the MUX for ALU source 2 to be from register B:
    \[ ALUSrcB = 00 \]
  - Set ALUcontrol to the func field operation:
    \[ ALUOp = 10 \]
Execute

- conditional branch
  - Set the MUX for ALU source 1 to be from register A:
    \[ ALUSrcA = 1 \]
  - Set the MUX for ALU source 2 to be from register B:
    \[ ALUSrcB = 00 \]
  - Set ALUcontrol to "-":
    \[ ALUOp = 01 \]
  - Set \( PCWriteCond \) signal which will update the PC if Zero is asserted
  - Set the MUX for input to the PC to be from ALUOut (holds the target address that was computed in the last cycle):
    \[ PCSource = 01 \]
    (note that the PC is written twice for taken conditional branches)

- jump
  - Set the MUX for input to the PC to be from the jump address:
    \[ PCSource = 10 \]
  - Set \( PCWrite \)

Data Memory Access & Register Write

Which control signals are generated depends on the opcode

- load
  - Set \( MemRead \)
  - Set the MUX so that the memory address comes from the ALU:
    \[ IorD = 1 \]

- store
  - Set \( MemWrite \)
  - Set the MUX so that the memory address comes from the ALU:
    \[ IorD = 1 \]
  - Where is the value that is to be written?

- R-type
  - Set the MUX to choose the rd field as the write register:
    \[ RegDst = 1 \]
  - Set \( RegWrite \)
  - Set the MUX to choose the ALU output as the data to write:
    \[ MemtoReg = 0 \]
Register Write from a Load

Which control signals are generated depends on the opcode

- **load**
  - Set the MUX to choose the rt field as the write register: \( \text{RegDst} = 0 \)
  - Set \( \text{RegWrite} \)
  - Set the MUX to choose the MDR as the data to write: \( \text{MemtoReg} = 1 \)