Novice Programmer

I CAN'T BELIEVE IT WORKED FIRST TIME!

DING!

Experienced Programmer

I CAN'T BELIEVE IT WORKED FIRST TIME...

DING!
Lecture 4: Basic Processor Architecture

Some slides adapted from Mary Jane Irwin at Penn State University for Computer Organization and Design, Patterson & Hennessy, © 2005
Review: CPU Performance

- execution time = seconds / program

\[
\frac{\text{Instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{Instruction}} \times \frac{\text{seconds}}{\text{cycle}}
\]

- programmer
- algorithms
- ISA
- compilers

- microarchitecture
- system architecture

- microarchitecture, pipeline depth
- circuit design
- technology
Review: Amdahl’s Law

- law of diminishing returns
  - make the common case faster
  - speedup = $\frac{\text{Perf}_{\text{new}}}{\text{Perf}_{\text{old}}} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{(1 - f) + \frac{f}{P}}$

- example
  - supposing floating point instructions are improved to run 2X but comprise only 10% of actual instructions, what’s the speedup?

\[
\begin{align*}
T_{\text{old}} &= (1 - f) + f \\
T_{\text{new}} &= (1 - f) + \frac{f}{P}
\end{align*}
\]
Review: CMOS Energy & Power Equations

\[ E = C_L V_{DD}^2 P_{0\rightarrow 1} + t_{sc} V_{DD} I_{peak} P_{0\rightarrow 1} + V_{DD} I_{leakage} \]

\[ f_{0\rightarrow 1} = P_{0\rightarrow 1} * f_{clock} \]

\[ P = C_L V_{DD}^2 f_{0\rightarrow 1} + t_{sc} V_{DD} I_{peak} f_{0\rightarrow 1} + V_{DD} I_{leakage} \]

- **Dynamic power**
  \(\approx 40 - 70%\) today and decreasing relatively

- **Short-circuit power**
  \(\approx 10\%\) today and decreasing absolutely

- **Leakage power**
  \(\approx 20 - 50\%\) today and increasing
our implementation of a processor is simplified
memory-reference instructions: \texttt{lw, sw}
arithmetic/logical instructions: \texttt{add, sub, and, or, slt}
branch instructions: \texttt{beq, j}

generic implementation
use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
decode the instruction (and read registers)
execute the instruction

all instructions (except \texttt{j}) use the ALU after reading the registers
Recap: Instruction Execution

- PC → instruction memory, fetch instruction
- register numbers → register file, read registers
- depending on instruction class
  - use ALU to calculate
    - arithmetic result
    - memory address for load/store
    - branch target address
- access data memory for load/store
- PC ← target address or PC + 4
Clocking Methodologies

- Clocking methodology defines when signals can be read and when they are written.
  - 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.
Overview (Simplified example circuit)
Fetching Instructions

- fetching instructions involves
  - reading the instruction from instruction memory
  - updating 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 register file
  - register file addresses are contained in the instruction
Executing R Format Operations

- R format operations (add, sub, slt, and, or)

<table>
<thead>
<tr>
<th>R-type:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

Perform the (op and funct) operation on values in rs and rt. Store the result back into register file (into location rd).

- Register file is not written every cycle (e.g. sw), so we need an explicit write control signal for register file.
Executing Load and Store Operations

- load and store operations involves

  compute memory address by adding the base register (read from register file during decode) to the 16-bit signed-extended offset field in the instruction

- store value (read from register file during decode) written to the data memory

- load value, read from data memory, written to register file
Executing Branch Operations

- branch operations involve:
  - compare the operands read from 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 replacing 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 instruction 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
Control signals for Fetch, Reg, and Memory

what determines the values needed on these control signals?
Adding the Control

- selecting the operations to perform (ALU, register file and memory read/write)
- controlling the flow of data (multiplexor inputs)

### Observations

- **op** field is always in bits 31-26
- Address 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
- Address 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
Single Cycle Datapath w/ Control Unit

**Instruction Memory**
- **Instruction Memory**: instr[31-0]
  - Instruction memory read address
  - instr[31-0]

**Control Unit**
- **Control Unit**: instr[31-26]
  - ALUOp
  - Branch
  - ALUSrc
  - RegDst
  - RegWrite

**Register File**
- **Register File**: instr[25-21]
  - Read addr 1
  - Read addr 2
  - Write addr
  - Write data
  - instr[15-11]
  - instr[15-11]

**Register Extension**
- **Register Extension**: instr[15-0]
  - Sign extend
  - 16
  - 32

**ALU**
- **ALU**: instr[5-0]
  - ALUOp
  - ALUSrc
  - RegWrite
  - zero
  - ovf

**Data Memory**
- **Data Memory**: read data
  - Write data
  - instr[5-0]

**Address Calculation**
- **Address Calculation**: instr[19-16]
  - instr[15-11]

**Instruction Processing**
- **Instruction Processing**: instr[15-11]
  - instr[19-15]
  - instr[5-0]

**ALU Control**
- **ALU Control**: Memory Read
  - MemWrite
  - MemtoReg
  - PC Src
  - Branch

**Branch**
- **Branch**: instr[5-0]
  - instr[5-0]

**Address Calculation**
- **Address Calculation**: instr[19-16]
  - instr[15-11]

**Instruction Processing**
- **Instruction Processing**: instr[15-11]
  - instr[19-15]
  - instr[5-0]
Branch Instruction Data/Control Flow
Adding the Jump Operation

- **control unit**
  - instr[31-26]
  - instr[25-21]
  - instr[20-16]
  - instr[15-11]
  - instr[15-0]
  - instr[5-0]

- **instruction memory**
  - read address
  - instr[31-0]

- **data memory**
  - read data
  - write data

- **register file**
  - read addr 1
  - read addr 2
  - write addr
  - read data 1
  - read data 2
  - write data

- **ALU**
  - ALUOp
  - RegDst
  - RegWrite
  - memtoReg
  - MemWrite
  - MemRead
  - PCSrc
  - ovf
  - zero
  - shift left 2
  - sign extend
  - ALUSrc

- **ALU control**
  - ALUOp
  - instr[31-26]
  - instr[25-21]
  - instr[20-16]
  - instr[15-11]
  - instr[15-0]
  - instr[5-0]
Instruction Type vs # of Required Cycles

- 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!**
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

![Diagram showing single cycle with lw, sw, and 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

- 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.
Multicycle Datapath Approach, con’t

- 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 b/w a pair of adjacent clock cycles (no write control signal needed)

**Diagram:**
- PC to memory (address)
- Memory to IR
- IR to MDR
- MDR to ALU
- ALU to ALUout
- ALUout to register file
- Memory to register file

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

Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)
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)
Multicycle Advantages & Disadvantages

- uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step

- multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles

- requires additional internal state registers, more muxes, and more complicated (FSM) control
Single Cycle vs. Multiple Cycle Timing

single cycle implementation:

\[ \text{cycle 1} \quad \text{cycle 2} \]

\[ \text{clk} \]

\[ \text{lw} \quad \text{sw} \quad \text{waste} \]

multiple cycle implementation:

\[ \text{cycle 1} \quad \text{cycle 2} \quad \text{cycle 3} \quad \text{cycle 4} \quad \text{cycle 5} \quad \text{cycle 6} \quad \text{cycle 7} \quad \text{cycle 8} \quad \text{cycle 9} \quad \text{cycle 10} \]

\[ \text{clk} \]

\[ \text{lw} \quad \text{sw} \]

\[ \begin{array}{cccccccccc}
\text{IFetch} & \text{Dec} & \text{Exec} & \text{Mem} & \text{WB} & \text{IFetch} & \text{Dec} & \text{Exec} & \text{Mem} & \text{IFetch}
\end{array} \]

multicycle clock slower than $1/5$th of single cycle clock due to state register overhead
A Hypothetical Data Path Timing Analysis

- **critical path**: a path through combinational circuit that takes as long or longer than any other.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>Fetch</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D Memory</th>
<th>R-Write</th>
<th>Total (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>

**clock cycle time**

\[= 4.7\text{ns} + t_{\text{setup}} + t_{\text{cq}}\]
A Hypothetical Data Path Timing Analysis

- **Critical path**: a path through combinational circuit that takes as long or longer than any other

**Clock cycle time**

\[ = 4.7\text{ns} + t_{\text{setup}} + t_{\text{cq}} \]

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>Fetch</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D Memory</th>
<th>R-Write</th>
<th>Total (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Cycle Time vs. CPI

- goal: balance amount of work done each cycle
  - load needs 5 cycles
  - add and store need 4
  - beq needs 3

<table>
<thead>
<tr>
<th></th>
<th>I Fetch</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D Memory</th>
<th>R-Write</th>
<th>Total (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Will multicycle design be faster?

- Let’s assume $t_{\text{setup}} + t_{\text{cq}}$ time for registers = 0.1 ns
  - Single cycle design:
    - Clock cycle time = $4.7 + 0.1 = 4.8$ ns
    - Time/inst = 1 cycle/inst × $4.8$ ns/cycle = 4.8 ns/inst
  - Multicycle design:
    - Clock cycle time = $1.0 + 0.1 = 1.1$
    - Time/inst = CPI × 1.1 ns/cycle (depends on the types or mixture of instructions!)

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>Fetch</th>
<th>Decode, R-Read</th>
<th>ALU</th>
<th>PC update</th>
<th>D Memory</th>
<th>R-Write</th>
<th>Total (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>-</td>
<td>.8</td>
<td>3.7</td>
</tr>
<tr>
<td>Load</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>.8</td>
<td>4.7</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>3.9</td>
</tr>
<tr>
<td>beq</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.9</td>
<td>.1</td>
<td>-</td>
<td>-</td>
<td>3.0</td>
</tr>
</tbody>
</table>
Will multicycle design be faster?

Let’s assume $t_{\text{setup}} + t_{\text{cq}}$ time for the register = 0.1 ns

- Single cycle design:
  - Clock cycle time = $4.7 + 0.1 = 4.8$ ns
  - Time/inst = 1 cycle/inst × 4.8 ns/cycle = 4.8 ns/inst

- Multicycle design:
  - Clock cycle time = $1.0 + 0.1 = 1.1$ ns/cycle
  - Time/inst = CPI × 1.1 ns/cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles needed</th>
<th>Instruction frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>10%</td>
</tr>
</tbody>
</table>

CPI

$= 4 \times 0.6 + 5 \times 0.2 + 4 \times 0.1 + 3 \times 0.1$

$= 2.4 + 1.0 + 0.4 + 0.3$

$= 4.1$
A More Extreme Example

- calculation assumptions:
  - most instructions take 10 nanoseconds (ns)
  - but multiply instruction takes 40ns
  - multiplies are 10% of all instructions

- how much faster is a multi-cycle data path over a single-cycle data path?
  - for simplicity, assume \( t_{\text{setup}} + t_{\text{cq}} = 0 \)
    - single-cycle \( \text{time/inst} = 40\text{ns} \)
    - multi-cycle with 10ns clock, \( \text{time/inst} = (1 \times 0.9 + 4 \times 0.1) \times 10 = 13\text{ns} \)
Announcement

• next lecture: pipeline (overview)
  ✓ Ch. 4.5 – 4.6 (HP1)

• MP assignment
  ✓ MP1 checkpoint due on 9/10 5pm