Why a Multiple Clock Cycle CPU?

- the problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- the solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- other advantages => reuse of functional units (e.g., alu, memory)

- ET = IC * CPI * CT

Breaking Execution Into Clock Cycles

- We will have five execution steps (not all instructions use all five)
  - fetch
  - decode & register fetch
  - execute
  - memory access
  - write-back
- We will use Register-Transfer-Language (RTL) to describe these steps
- Introduces extra registers when:
  - signal is computed in one clock cycle and used in another, AND
  - the inputs to the functional block that outputs this signal can change before the signal is written into a state element.
- Significantly complicates control. Why?
- The goal is to balance the amount of work done each cycle.
1. Fetch

IR = Mem[PC]
PC = PC + 4

(may not be final value of PC)

2. Instruction Decode and Register Fetch

A = Reg[IR[25-21]]
B = Reg[IR[20-16]]
ALUOut = PC + (sign-extend (IR[15-0]) << 2)

• compute target before we know if it will be used (may not be branch, branch may not be taken)
• ALUOut is a new state element (temp register)
• everything up to this point must be Instruction-independent, because we still haven’t decoded the instruction.
• everything instruction (opcode)-dependent from here on.

3. Execution, memory address computation, or branch completion

• Memory reference (load or store)
  ALUOut = A + sign-extend(IR[15-0])
• R-type
  ALUout = A op B
• Branch
  if (A == B)  PC = ALUOut

At this point, Branch is complete, and we start over; others require more cycles.
4. Memory access or R-type completion

- Memory reference
  - load
    \[ MDR = \text{Mem}[\text{ALUout}] \]
  - store
    \[ \text{Mem}[\text{ALUout}] = B \]
- R-type
  \[ \text{Reg}[\text{IR}[15-11]] = \text{ALUout} \]

R-type is complete, store is complete.

5. Memory Write-Back

\[ \text{Reg}[\text{IR}[20-16]] = \text{MDR} \]

*load is complete*

---

**Summary of execution steps**

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>( \text{IR} = \text{Mem}[\text{PC}] ) |  ( \text{PC} = \text{PC} + 4 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/ register fetch</td>
<td>( A = \text{Reg}[\text{IR}[25-21]] ) |  ( B = \text{Reg}[\text{IR}[20-16]] ) |  ( \text{ALUout} = \text{PC} + \text{sign-extend}(\text{IR}[15-0]) \ll 2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch completion</td>
<td>( \text{ALUout} = A \op B ) |  ( \text{ALUout} = A + \text{sign-extend}(\text{IR}[15-0]) ) |  ( \text{if}(A = B) \text{ then } \text{PC} = \text{ALUout} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>( \text{Reg}[\text{IR}[15-11]] = \text{ALUout} ) |  ( \text{memory-data} = \text{Mem}[\text{ALUout}] ) |  ( \text{or } \text{Mem}[\text{ALUout}] = B )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>( \text{Reg}[\text{IR}[20-16]] = \text{memory-data} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Complete Multicycle Datapath**

(support for what instruction just got added?)
1. Instruction Fetch

IR = Memory[PC]
PC = PC + 4

2. Instruction Decode and Reg Fetch

A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUOut = PC + (sign-extend(IR[15-0]) << 2)

3. Execution (R-type)

ALUout = A op B

4. R-type Completion

Reg[IR[15-11]] = ALUout
3. Branch Completion

if (A == B)  PC = ALUOut

4. Memory Access (Load)

memory-data = Memory[ALUout]

3. Memory Address Computation

ALUout = A + sign-extend(IR[15-0])

4. Memory Access (Store)

CSE 141 Memory[ALUout] = B

Dean Tullsen
5. Load Write-back

![Diagram of Load Write-back](image)

- Register \[\text{IR[20-16]}\] = memory-data

3. JMP Completion

![Diagram of JMP Completion](image)

- Program Counter (PC) = PC[31-28] | (IR[25-0] <<2)

Multicycle Control

- Single-cycle control used combinational logic
- Multi-cycle control uses ??
- FSM defines a succession of states, transitions between states (based on inputs), and outputs (based on state)
- First two states same for every instruction, next state depends on opcode

Multicycle Control FSM

![Diagram of Multicycle Control FSM](image)
First two states of the FSM

Instruction Fetch, state 0
Instruction Decode/Register Fetch, state 1

MemRead
ALUSrcA = 0
for D = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Memory Inst
R-type Inst
Branch Inst
Jump Inst
FSM
FSM
FSM

R-type Instructions

BEQ Instruction

from state 1

Execution

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Completion

? 

To state 0

A = Register(IR[25-21])
B = Register(IR[20-16])
Target = PC + (sign-extend (IR[15-0]) << 2)

from state 1

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

To state 0

CSE 141
Dean Tullsen
### Memory Instructions

- **from state 1**
- **Address Computation**
  - **MemRead**: IorD = 1
  - **MemWrite**: IorD = 1
- **write-back**
  - **MemRead**: MemRef = 1
  - **MemWrite**: MemRef = 1

- **To state 0**

### JMP Instruction

- **from state 1**
- **PCWrite**
  - **PCSource**: 10

- **To state 0**

### The Whole FSM

- Instruction decode/register fetch
  - **Instruction fetch**
    - **lw**: $t2, 0($t3)
    - **lw**: $t3, 4($t3)
    - **beq**: $t2, $t3, Label #assume not taken
    - **add**: $t5, $t2, $t3
    - **sw**: $t5, 8($t3)
    - **Label**: ...

- **Branch completion**
  - **Branch**
    - **lw**: $t2, 0($t3)
    - **lw**: $t3, 4($t3)
    - **beq**: $t2, $t3, Label #assume not taken
    - **add**: $t5, $t2, $t3
    - **sw**: $t5, 8($t3)
    - **Label**: ...

- **Memory access computation**
  - **MemRead**: IorD = 1
  - **MemWrite**: IorD = 1
  - **MemRef = 1**

- **Execution**

- **Jump completion**

- **R-type completion**

- **T-type completion**

### Some Questions

- How many cycles will it take to execute this code?
  ```
  lw $t2, 0($t3)
  lw $t3, 4($t3)
  beq $t2, $t3, Label #assume not taken
  add $t5, $t2, $t3
  sw $t5, 8($t3)
  Label: ...
  ```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of $t2 and $t3 take place?

- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?
Finite State Machine for Control

- Implementation:
  - Datapath control outputs
  - Combinational control logic
  - Outputs
  - Inputs
  - State register
  - Inputs from instruction register opcode field
  - Next state

ROM Implementation

- ROM = "Read Only Memory"
  - Values of memory locations are fixed ahead of time
  - A ROM can be used to implement a truth table
  - If the address is m-bits, we can address $2^m$ entries in the ROM.
  - Our outputs are the bits of data that the address points to.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 1</td>
<td>0 0 1 1</td>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
<td>0 1 1 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
<td>0 1 1 0</td>
<td>0 1 1 1</td>
<td>1 0 0 0</td>
<td>1 0 0 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 0 1</td>
<td>1 0 1 0</td>
<td>1 0 1 1</td>
<td>1 1 0 0</td>
<td>1 1 0 1</td>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 1 0</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- How many inputs are there?
  - 6 bits for opcode, 4 bits for state = 10 address lines
    (i.e., $2^{10} = 1024$ different addresses)

- How many outputs are there?
  - 16 datapath-control outputs, 4 state bits = 20 outputs

- ROM is $2^{10} \times 20 = 20K$ bits (and a rather unusual size)

- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

Multicycle CPU Key Points

- Performance gain achieved from variable-length instructions
- $ET = IC \times CPI \times$ cycle time
- Required very few new state elements
- More, and more complex, control signals
- Control requires FSM
Exceptions

• There are two sources of non-sequential control flow in a processor
  – explicit branch and jump instructions
  – exceptions

• Branches are synchronous and deterministic
• Exceptions are typically asynchronous and non-deterministic

Guess which is more difficult to handle?

(control flow refers to the movement of the program counter through memory)

Exceptions and Interrupts

the terminology is not consistent, but we’ll refer to:
• exceptions as any unexpected change in control flow
• interrupts as any externally-caused exception

So then, what is:
  – arithmetic overflow
  – divide by zero
  – I/O device signals completion to CPU
  – user program invokes the OS
  – memory parity error
  – illegal instruction
  – timer signal

For now...

• The machine we’ve been designing in class can generate two types of exceptions.
  – arithmetic overflow
  – illegal instruction

• On an exception, we need to
  – save the PC (invisible to user code)
  – record the nature of the exception/interrupt
  – transfer control to OS
Handling exceptions

• PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
• A status register, and a single exception handler may be used to record the exception and transfer control, or
• A vectored interrupt transfers control to a different location for each possible type of interrupt/exception

Supporting exceptions

• For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user’s PC
  - Cause: A register to record the cause of the exception
    - we’ll assume undefined inst = 0, overflow = 1
• We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
• We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.

Supporting exceptions in our DataPath

Supporting exceptions in our FSM
Key Point

- Exception-handling is difficult in the CPU, because the interactions between the executing instructions and the interrupt are complex and sometimes unpredictable.