Multiple Clock Cycle CPU

or

Breaking Up Is Hard To Do

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

Breaking Execution Into Clock Cycles

- 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)
  \[ ALUOut = A + \text{sign-extend}(IR[15-0]) \]
- R-type
  \[ ALUOut = A \text{ op } B \]
- Branch
  \[ \text{if } (A == B) \text{ PC = ALUOut} \]

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

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

R-type is complete, store is complete.

5. Memory Write-Back

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

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

---

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

3. Memory Address Computation

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

4. Memory Access (Load)

memory-data = Memory[ALUout]

4. Memory Access (Store)

Memory[ALUout] = B
5. Load Write-back

Reg[IR[20-16]] = memory-data

3. JMP Completion

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

start

Instruction fetch

Decode and Register Fetch

Memory instructions
R-type instructions
Branch instructions
Jump instruction
First two states of the FSM

Instruction Fetch, state 0
- MemRead
- ALUSrcA = 0
- IorD = 0
- IRWrite
- ALUSrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction Decode/ Register Fetch, state 1
- ?
- Opcode = LW or SW
- Opcode = R-type
- Opcode = BEQ

R-type Instructions

from state 1
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 10

To state 0

BEQ Instruction

from state 1
- ALUSrcA = 1
- ALUSrcB = 00
- ALUOp = 01
- PCWriteCond
- PCSource = 01

To state 0

Memory Instructions

from state 1
- MemRead
- IorD = 1
- MemWrite
- MemtoReg = 1
- RegDst = 0

To state 0
JMP Instruction

from state 1

PCWrite
PCSource = 10

To state 0

The Whole FSM

Finite State Machine for Control

• Implementation:

Microcode

• Replace the next-state logic with a microsequencer (mini microcontroller), and...
• Implement the control logic truth table as a ROM, and think of each line in the ROM (which specifies a set of control signals for a single cycle) as an “instruction”.
• This gives you microcode. You can now specify the control logic for a new machine by writing microcode.
• In many ways, this was how CISC machines got out of hand – too easy to add new, complex instructions.
**Multicycle CPU Key Points**

- Performance gain achieved from variable-length instructions
- ET = IC * CPI * 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
  - 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

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 FSM

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

ALUSrcA = 0
ALUsrcB = 11
ALUOp = 00

from state 1

R-type Inst
Memory Inst
Branch Inst
Jump Inst

ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Overflow
To state 11

Illegal Instruction
To state 0

Interrupt Handler

Sub

EPC
PC

To state 10

PCWrite
EPCWrite

IntCause=1
CauseWrite

IntCause=0
CauseWrite

ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource=11

Key Point

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