Csci 136 Computer Architecture II
*– Designing a Multi-Cycle Processor*

*Xiuzhen Cheng*
*cheng@gwu.edu*

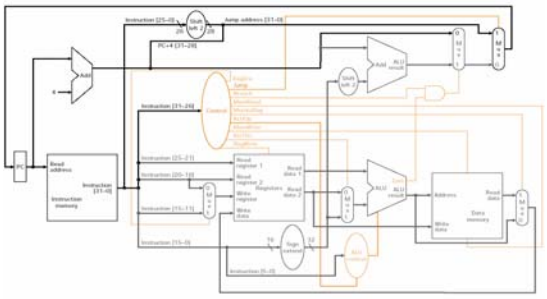
## Announcement

- **Homework assignment #8, Due time – before class, March 29.**
  - Readings: Sections 5.5-5.6
  - Problems: 5.32, 5.33, 5.36.
- **Quiz #3: March 29**
- **Project #3 is on-line**

## Review on Single Cycle Datapath

- **Subset of the core MIPS ISA**
  - Arithmetic/Logic instructions: AND, OR, ADD, SUB, SLT
  - Data Flow instructions: LW, SW
  - Branch instructions: BEQ, J
- **Five steps in processor design**
  - Analyze the instruction
  - Determine the datapath components
  - Assembly the components
  - Determine the control
  - Design the control unit

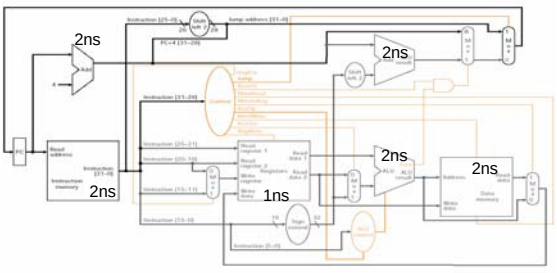
## The Complete Single Cycle Datapath



How lw, sw, R-Type, beq, j instructions work?

Why the design of AUL control takes two levels?

## Delays in Single Cycle Datapath



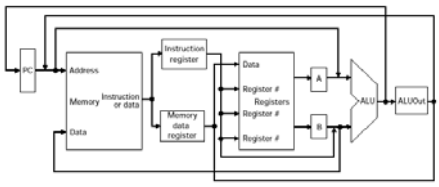

What are the delays for lw, sw, R-Type, beq, j instructions?

## Remarks on Single Cycle Datapath

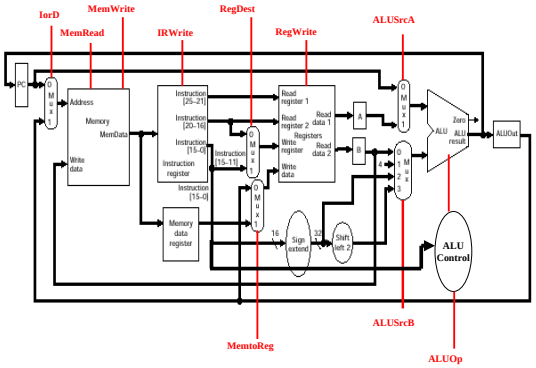
- **Single Cycle Datapath ensures the execution of any instruction within one clock cycle**
  - Functional units must be duplicated if used multiple times by one instruction. E.g. ALU. Why?
  - Functional units can be shared if used by different instructions
- **Single cycle datapath is not efficient in time**
  - Clock Cycle time is determined by the instruction taking the longest time. Eg. lw in MIPS
  - Variable clock cycle time is too complicated.
  - Multiple clock cycles per instruction – this lecture
  - Pipelining – Chap 6

## Multiple Cycle Datapath

- **Minimizes Hardware: 1 memory for data and instruction, 1 ALU**
  - A functional unit can used more than once as long as it is used on different clock cycles
  - **Advantages:** shared functional units, different cycles for different instructions, short clock cycles
  - **Assumptions:** each clock cycle can accommodate at most one of the following operations: a memory access, a register file access, or an ALU
    - **Temporary registers:** A, B, IR, MDR, ALUOut
- **A high level view of the multi-cycle datapath**



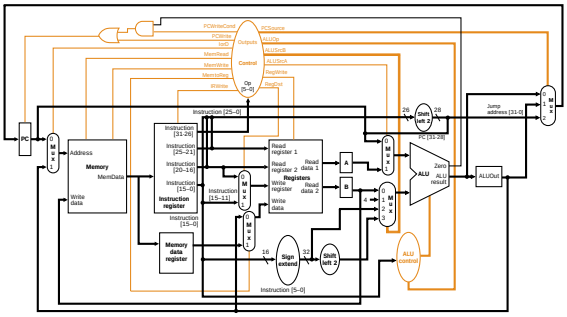

## Multiple Cycle Datapath with Control




## Supporting Jump and Conditional Branch

- **Need PC control**
  - PC is updated conditionally for branch and unconditionally for normal increment and jumps
  - Control unit generates **PCWrite** and **PCWriteCond** based on op code of the instruction
    - For branch, PCWriteCond and Zero must be set
    - For Jump or other unconditional PC update, PCWrite must be set
    - Thus PCControl = (PCWriteCond and Zero) or PCWrite
  - PC source selection
    - A mux with 3 inputs: PC+4, PC + signExt(IR[15-0])<<2), PC[31-28] || (IR[25-0]<<2)

## The Complete Multicycle Dataath



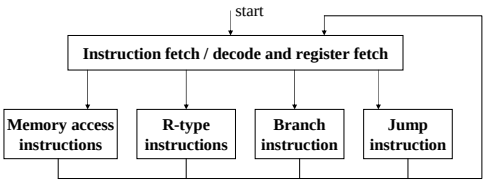

## Breaking Instruction Execution into Multiple Cycles

- **Instruction Fetch**
  - IR = Memory[PC]  PC = PC+4
- **Instruction Decode/Register Fetch**
  - A = Reg[IR[25-21]]  B = Reg[IR[20-16]]
    ALUOut = PC + (sign-extend(IR[15-0])<<2)
- **Execution, Address compuation, branch/jump completion**
  - R-type: ALUOut = A op B
  - Memory access: ALUOut = A + sign-extend(IR[15-0])
  - Branch: if (A==B) then PC = ALUOut
  - Jump: PC = PC[31-28] || (IR[25-0]<<2)
- **Memory Access or R-type Completion**
  - R-type: Reg[IR[15-11]] = ALUOut
  - Load: MDR = Memory[ALUOut] or
    Store: Memory[ALUOut] = B
- **Memory read completion**
  - Load: Reg[IR[20-16]] = MDR

## Defining the Control

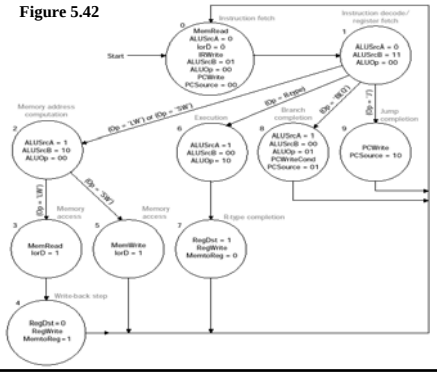
- **The control of the multicycle datapath must specify both the signals to be set in any step and the next step in the sequence**
- **Two techniques:**
  - **Finite state machine**
    - **Each state (a circle) contains the valid control signals**
    - **Directional links point to next state**
    - **Each cycle corresponds to one state**
    - **FSM is the graphical representation of the control**
  - **Microprogramming**
    - **Assume the set of control signals that must be asserted in a state as an instruction to be executed by the datapath**
    - **Microprogram is a symbolic representation of the control that will be translated by a program to control logic**

## The high-level view of the FSM control



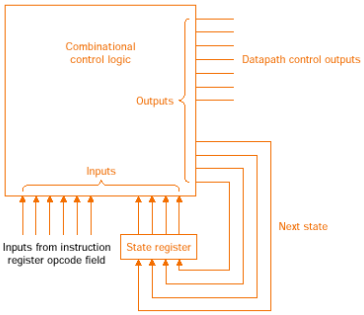

## Finite State Machine Control

Figure 5.42



## Implementation of the FSM Control

Figure 5.37




## In-Class Exercise

- How to modify the complete datapath (Figure 5.33) for the multiple cycle implementation to accommodate the `ori` (or immediate) instruction? Given the FSM control design
- What about `addi`, `jal` and `jr` instructions?

## Exception Handling (1/6)

- Exception vs. Interrupt: both are unexpected events in control flow
  - Interrupt: externally caused events
  - Exception: internally caused events
- Consider two types of exceptions in our current implementation
  - Arithmetic overflow
  - Undefined instructions
- How exceptions are handled?
  - Save the address of the offending instruction to **EPC**. How to find out the address of the offending instruction?
  - Transfer control to the OS at some specified address. How?
  - May transfer control back through **EPC**

## Exception Handling (2/6)

- `Cause register`: a status register to record the reason of the exception
  - One single entry point for all exceptions can be used.
- `Vectored interrupt`
  - Control will be transferred based on the cause of the exception
  - Eg:

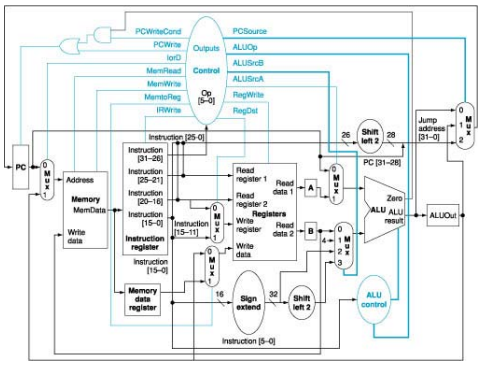
| Exception Type | Exception Vector Address |
|---|---|
| `undefined instruction` | `C000 0000` |
| `arithmetic overflow` | `C000 0020` |

example

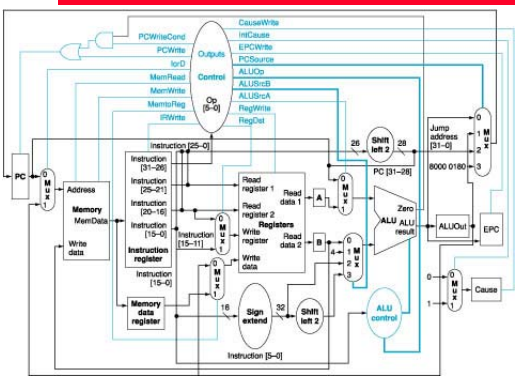
## Exception Handling (3/6)

- `EPCWrite` and `CauseWrite`
- `IntCause`
  - The LSB of the cause register to hold the state
  - 0 for undefined instruction
  - 1 for arithmetic overflow
  - `IntCause` will be used to set the LSB of the `Cause` register
- `exception address: 8000 0180`
  - The entry point for exceptions
- **Question: How to modify the current datapath for exception handling?**

## Exception Handling (4/6)



## Exception Handling (5/6)



## Exception Handling (6/6)

## Questions?