ECE 485/585
Computer Organization and Design

Lecture 9: Multicycle Datapath Design
Fall 2022

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

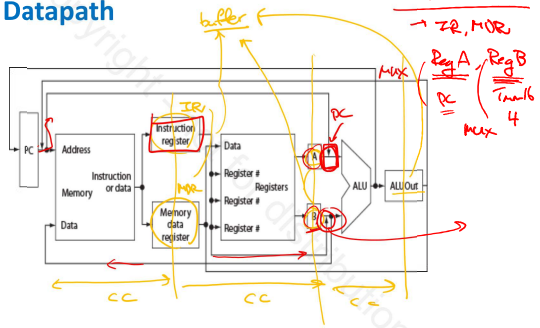
Multicycle Implementation

- Multicycle Implementation:
  - Break each instruction into series of steps corresponding to functional unit operations
  - *Each* step in execution will take 1 clock cycle
  - Instructions can take different numbers of clock cycles to execute
- Multicycle implementation allows a functional unit to be used more than once per instruction
  - As long as it is used on different clock cycles

Comparison with Single Cycle Datapath

- Datapath implementation differences against single cycle datapath:
  - A single memory is used for both instruction and data
  - There is single ALU
  - More registers are added in order to hold the output of functional units until the value is used in a subsequent cycle

Abstract Look at the Multicycle Datapath



State Elements

- Data storage:
  - Programmer visible state elements:
    - Data used in subsequent instructions will be stored in register file, PC or memory
  - Additional Registers
    - Data used by the same instruction in a later cycle must be stored into additional registers

Determine the Registers

- How do we determine where to put additional registers?
  - Which combinational units will fit in one clock cycle?
  - What data are needed in later cycles implementing instruction?
- Assume that a clock cycle can accommodate at most one operation:
  - Memory access
  - Register file access (2 reads and 1 write)
  - ALU operation
- Need to save the data produced by any of these units into temporary registers for use on a later clock cycle

Temporary Registers

- For MIPS implementation, following registers are added:
  - Instruction Register (IR) and Memory data register (MDR)
  - A and B registers are used to hold the register file read outputs
  - ALUOut register hold the output of ALU
- All registers except IR hold the data between a pair of clock cycles
- IR register needs to retain the instruction data until the end of execution
  - requires a write control signal

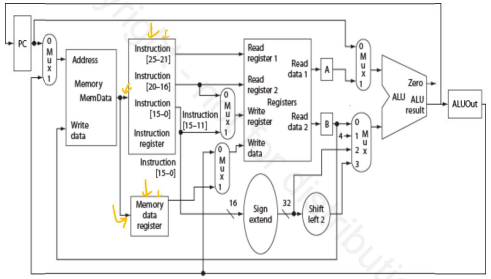
Datapath Changes

- Our goal: Reduce memory units to 1 and eliminate 2 adders
- Sharing same functional units indicate:
  - More multiplexors and/or more inputs to existing multiplexors.
- Functional Unit Memory:
  - A multiplexor is needed to select between two sources:
    - PC for instruction access
    - ALUOut for data access

ALU Multiplexor Design

- Functional Unit ALU:
  - Three ALUs are replaced by a single ALU. Two changes to the datapath are required:
  - An additional multiplexor is added for the first ALU input. Select between A register and PC
  - The multiplexor on the second ALU input is now 4-way multiplexor:
    - B Register
    - Constant 4 (increment PC)
    - Sign extended offset field
    - Sign extended and shifted offset field (Branch)

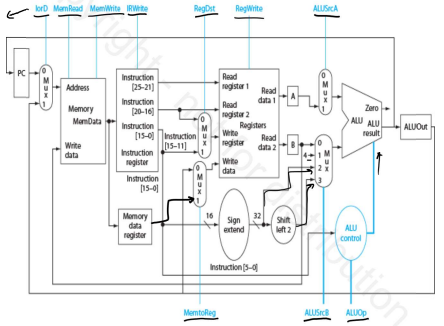
New Multicycle Datapath



Control Lines

- Different set of control signals compared to single cycle implementation
- Programmer visible state units need write control signal
  - Memory, PC, register file and IR
- Memory needs a read signal
- Two input multiplexors need single control line
- Four input multiplexor needs two control signals
- Same ALU control unit from the single cycle implementation can be used for ALU in the multicycle implementation

Multicycle Datapath with Control



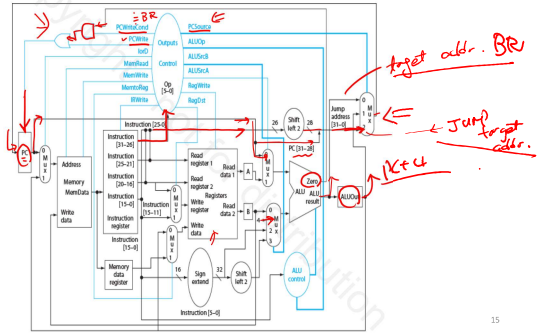
Branch and Jump Datapath

- For jump and branch instruction implementation, there are three possible sources for the value to be written to PC:
  - The output of ALU which is PC+4
  - The register ALUOut – stores the address of the branch target after it is computed
  - The lower 26 bits of IR shifted left by two and concatenated with upper 4 bits of the incremented PC (for jump instruction)

Control Signals for Branch and Jump

- PC is written either conditionally or unconditionally:
  - For normal increment and jump instructions; it is written unconditionally – use PCWrite control signal
  - For a conditional branch such as beq, the incremented value is replaced with value in ALUOut if comparison result is true. – use PCWriteCond control signal
- The write signal for the PC is implemented by a small logic that combines PCWrite, PCWriteCond and Zero signals

Complete Multicycle Datapath



Breaking the Instruction Execution into Clock Cycles

- What happens in each clock cycle of the multicycle execution?
- Maximize performance by breaking execution of any instruction into series of steps
  - Keep the amount of work per cycle equal
  - Clock cycle is determined by the slowest step!
- All operations listed in one step occur in parallel within 1 clock cycle.
- Successive steps occur operate in series in different clock cycles
- Limitation of one ALU operation, one Memory access and one register file access determines what can fit in one step

5 Steps for MIPS Datapath

- Each MIPS instruction needs three to five steps:
  - Instruction Fetch (IF)
  - Instruction Decode and register fetch (ID)
  - Execution, Memory address computation and branch completion (EX)
  - Memory Access or R-type instruction completion step (MEM)
  - Memory Read completion step (WB)

Step 1: Instruction Fetch (IF)

- Fetch instruction from memory and compute the address of the next sequential instruction:

  IR <=Memory[PC]

  PC <=PC+4

- Operations:
  - Send PC to memory as address, perform a read and write the instruction to IR. Increment PC by 4
- The increment of PC and instruction memory access occur in parallel

Control Signals for IF

- Control Signals:
  - Fetch instruction:
    - Assert MemRead and IRWrite
    - Set IorD to 0 (select PC as the source for memory)
  - Increment PC by 4:
    - Set ALUSrcA signal to 0 (send PC to ALU)
    - Set ALUSrcB signal to 01 (send 4 to ALU)
    - Set ALUOp to 00 (ALU operation add)
  - Store incremented instruction address back to PC
    - Set PCSource to 00
    - Set PCWrite

Step 2: Instruction Decode and Register Fetch (ID)

- Instruction is still not known, but we can perform actions that are not harmful:
- Read two registers and store them in temporary registers; compute branch target address with ALU

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

- Operations:
  - Access register file to read rs and rt
  - Store them in registers A and B
  - Compute branch target address
  - Store the address in ALUOut (it will be used in the next clock cycle if instruction is a branch)

Control Signals for ID

- Control Signals
  - Branch target address calculation
    - Set ALUSrcA to 0 (PC is sent to ALU)
    - Set ALUSrcB to 11 (sign extended and shifted offset field is sent to ALU)
    - Set ALUOp to 00 (ALU operation is addition)
  - A and B are written on every clock cycle no control signals required

Step 3: Execution, Memory Address Computation, Branch Completion (EX)

- First two steps are same for all instructions
- For step 3 and rest of the steps, datapath operation is determined by the instruction class
- Four different functions depending on the class during the step 3:
  - Memory Reference
  - Arithmetic Logical Instruction (R-Type)
  - Branch
  - Jump

Memory Reference for EX

- Memory reference instruction:

```
ALUOut <= A + sign-extend (IR[15:0]);
```

- Operation:
  - ALU is adding the operands, temporary register A and sign extended immediate field; to form the memory address
- Control Signals:
  - Set ALUSrcA=1 (first ALU input is register A)
  - Set ALUSrcB=10 (second ALU input is sign-extended immediate field)
  - Set ALUOp=00 (ALU operation is addition)

R-Type Instruction for EX

- Arithmetic-logical instruction:

```
ALUOut <= A op B;
```

- Operation:
  - ALU is performing the operation specified by the function code on two values read from registers A and B in the previous cycle
- Control Signals:
  - Set ALUSrcA=1 (Register A is the first ALU input)
  - Set ALUSrcB=00 (Register B is the second ALU input)
  - Set ALUOp=10 (funct field is used to determine the ALU control signal settings)

Branch Instruction for EX

- Branch Instruction

```
If (A==B)  PC <=ALUOut;
```

- Operation:
  - ALU is used to do the equal comparison between two registers read in the previous clock cycle
  - Zero signal output of ALU will be used to determine whether to branch or not
- Control Signals:
  - Set ALUSrcA=1
  - Set ALUSrcB=00
  - Set ALUOp=01 (ALU operation is subtraction)
  - Assert PCWriteCond if Zero output is asserted
  - Set PCSource=01 (The value written into PC comes from ALUOut which holds the branch target address from previous cycle)

Jump Instruction for EX

- Jump instruction
  - PC is replaced by the jump address which is the lower 26 bits of IR shifted left by two and concatenated with upper 4 bits of the incremented PC
- Control Signals:
  - PCSource = 10
  - PCWrite is asserted (write jump address to PC)

Step 4: Memory Access or R-Type Completion (MEM)

- Depending on the instruction type, two different operations
  - A load or store instruction accesses memory
  - An arithmetic-logical instruction writes the result
- For a load instruction, a value is retrieved from memory and stored into memory data register (MDR). It will be used in the next clock cycle

Memory Access for MEM

- Memory reference:
  MDR <= Memory [ALUOut] for load or
  Memory[ALUOut] <=B for store
- Operation:
  - If instruction is load, a data word is retrieved from memory and written to MDR
  - If instruction is store, data is written into memory
- Control signals:
  - Assert MemRead for a load
  - Assert MemWrite for a store
  - Set IorD =1 (memory address comes from ALU)

R-Type Instruction for MEM

- Arithmetic-logical instruction result:
  Reg[IR[15:11]] <= ALUOut
- Operation:
  - Place the contents of the ALUOut into Result register. ALUOut holds the result of ALU operation from previous cycle
- Control Signals:
  - Set RegDst=1 (use bits 15:11 for write destination)
  - Assert RegWrite
  - Set MemtoReg=0 (output of ALU is written not memory unit)

Step 5: Memory Read Completion (WB)

- Load instruction completes by writing back the value from the memory
  Reg[IR[20:16]]<=MDR
- Operation:
  - Write the load data (stored in MDR in the previous cycle) into register file
- Control Signals:
  - Set MemtoReg=1 (write result from memory)
  - Assert RegWrite (write to register file)
  - Set RegDst=0 (select rt (bits 20:16) for register number)

Execution Steps Summary

Step name	Action for R-type instructions	Action for memory reference	Action for branches	Action for Jumps
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 computation, branch, jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A==B) PC <=ALUOut	PC<=(PC[31:28],IR[25:0],00)
Memory access or R-type completion	Reg[IR[15:11]] <= ALUOut	MDR <= Memory [ALUOut] or Memory[ALUOut] <=B		
Memory read completion		Reg[IR[20:16]]<= MDR		

CPI in a Multicycle CPU – Example

- Using the SPECINT2000 instruction mix, what is the CPI assuming each state in multicycle CPU requires 1 clock cycle?
- 25% loads, 10% stores, 11% branches, 2% jumps and 52% ALU
- Number of clock cycles for each instruction is
  - Loads=5
  - Stores=4
  - ALU=4
  - Branches=3
  - Jump=3

$CPI_{overall} = \sum \frac{IC_i}{IC_{total}} \times CPI_i$

$= 0.25 \times 5 + 0.1 \times 4 + 0.11 \times 3 + 0.02 \times 3 + 0.52 \times 4$

$\doteq 4.12$

CPI in a Multicycle CPU – Example

- CPI can be found by:

$$CPI = \frac{CPU\ clock\ cycles}{Instruction\ count} = \frac{\sum Instruction\ count_i \times CPI_i}{Instruction\ count}$$

$$CPI = \sum \frac{Instruction\ count_i}{Instruction\ count} \times CPI_i$$

where $\frac{Instruction\ count_i}{Instruction\ count}$ is instruction frequency for class *i*

Hence: CPI=0.25x5 + 0.10x4 + 0.52x4 + 0.11x3 + 0.02x3 = 4.12

Worst case CPI was 5.0