Multi cycle –
Breaking instructions into steps

Breaking instructions into steps

- Our goal is to break up the instructions into *steps* so that
  - **each step takes one clock cycle**
  - each cycle uses at most **once each major functional unit** so that such units do not have to be replicated
  - functional units can be **shared** between different cycles within one instruction
- **Data at end of one cycle to be used in next *must be stored* !!**

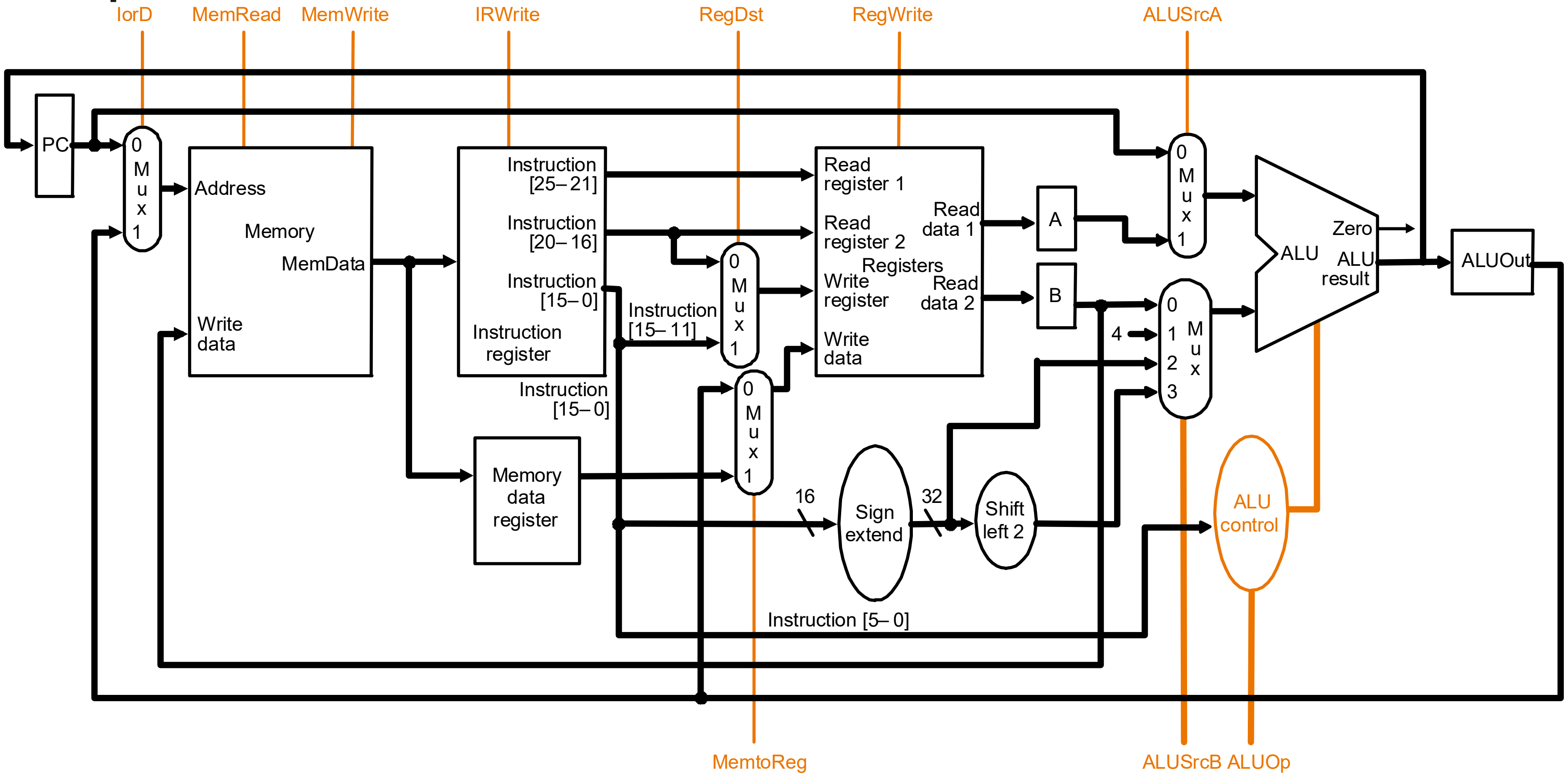
- Multi-cycle implementation attempt to keep the amount of work per cycle roughly equal.
  - Restrict each step to contain at most
    - One ALU operation
    - One register file access
    - One memory access
- With this restriction clock cycle could be as short as the longest of these operations.

Breaking instructions into steps

- We break instructions into the following *potential* execution steps ;each step takes one clock cycle
  1. Instruction fetch and PC increment (**IF**)
  2. Instruction decode and register fetch (**ID**)
  3. Execution, memory address computation, or branch completion (**EX**)
  4. Memory access or R-type instruction completion (**MEM**)
  5. Writing data back to register file(**WB**)

- Each MIPS instruction takes from 3 – 5 cycles (steps)

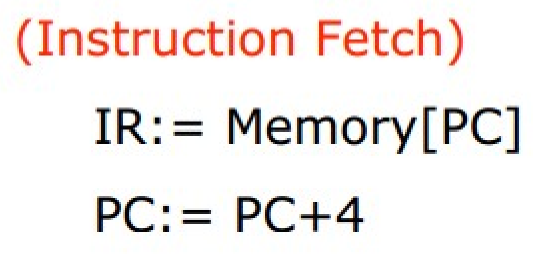
Multicycle Datapath with Control



… with control lines and the ALU control block added – *not all* control lines are shown

Step 1: Instruction Fetch & PC Increment (**IF**)

- Use PC to get instruction and put it in the instruction register. Increment the PC by 4 and put the result back in the PC.



```
(Instruction Fetch)
    IR:= Memory[PC]
    PC:= PC+4
```

Step 2: Instruction Decode and Register Fetch (**ID**)

- Read registers **rs** and **rt** in case we need them.
  Compute the branch address in case the instruction is a branch.

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

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

- ALU performs one of four functions *depending* on instruction type
  - memory reference:
    ALUOut = A + sign-extend(IR[15-0]);
  - R-type:
    ALUOut = A op B;
  - branch (instruction *completes*):
    if (A==B) PC = ALUOut;
  - jump (instruction *completes*):
    PC = PC[31-28] || (IR(25-0) << 2)

Step 4: Memory access or R-type Instruction Completion (MEM)

- Again *depending* on instruction type:
- Loads and stores access memory
  - load

    MDR = Memory[ALUOut];
  - store (instruction *completes*)

    Memory[ALUOut] = B;

- R-type (instructions *completes*)
  Reg[IR[15-11]] = ALUOut;

Step 5: Memory Read Completion (**WB**)

- Again *depending* on instruction type:
- Load writes back (instruction *completes*)

Reg[IR[20-16]]= MDR;

Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing
Reg[IR[20-16]]= Memory[ALUOut]; for loads in Step 4. This would eliminate the MDR as well.

The **reason** this is not done is that, to keep steps balanced in length, the design restriction is to allow **each step to contain *at most* one ALU operation, or one register access, or one memory access.**

Summary of Instruction Execution

Step	**Step name**	**Action for R-type instructions**	**Action for memory-reference instructions**	**Action for branches**	**Action for jumps**
1: IF	Instruction fetch	IR = Memory[PC] PC = PC + 4			
2: ID	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
3: EX	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
4: MEM	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		