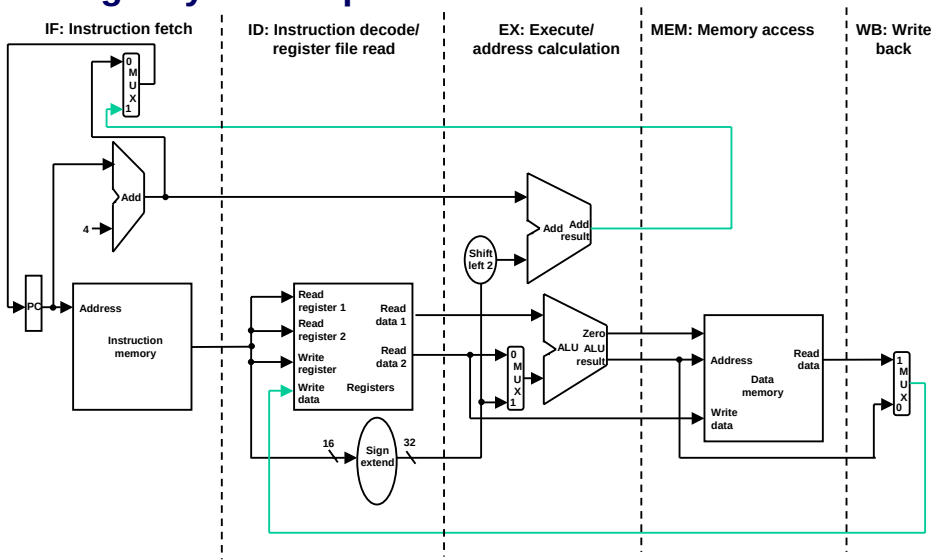
L11: Pipelining

Pipelining the datapath

- Recall the single cycle and multicycle datapath execution can be divided into 5 stages:
  1. IF: Instruction fetch
  2. ID: Instruction decode and register file read
  3. EX: Execution or address calculation
  4. MEM: Data memory access
  5. WB: Write back
- This division naturally leads to a five stage pipeline, which means up to five instructions are processed during a single clock cycle

Single cycle datapath

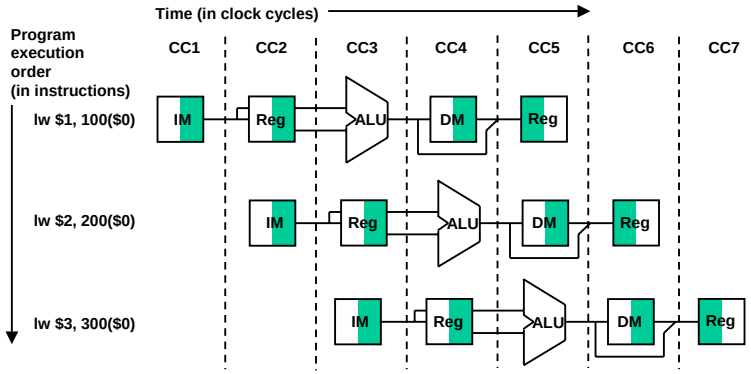


Notes on the Single cycle datapath

- Each processing step can be mapped onto the datapath from left to right
  - Exceptions are the PC update step, which sends the ALU result to the left, and
  - The write back step, which sends data from memory left to be written to the register file
- Data flowing from *right to left* does not affect the current instruction; only later instructions in the pipeline are influenced by these reverse data movements, which can lead to control and data hazards, respectively.

Towards pipelining…

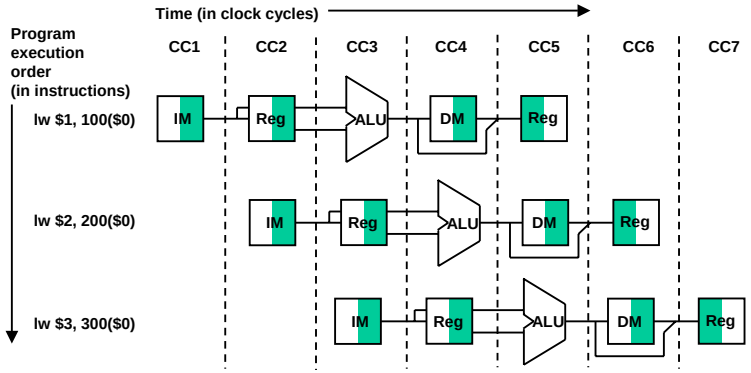
- We can see what happens in pipelined execution by pretending each instruction has its own datapath, and placing each of these datapaths on a timeline to show their timing relationship.



Towards pipelining…

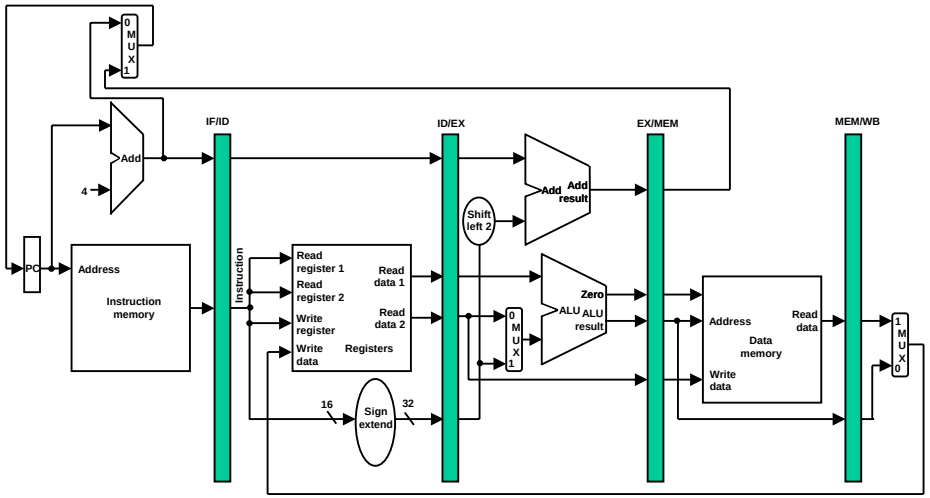
- Like when we examined the multicycle datapath, we add registers to hold data so that portions of the datapath can be shared during instruction execution

Pipelined execution of the single cycle datapath



- Resources such as IM are used in only one of the five execution stages and can therefore be shared by other instructions during the other four stages

Pipelined datapath

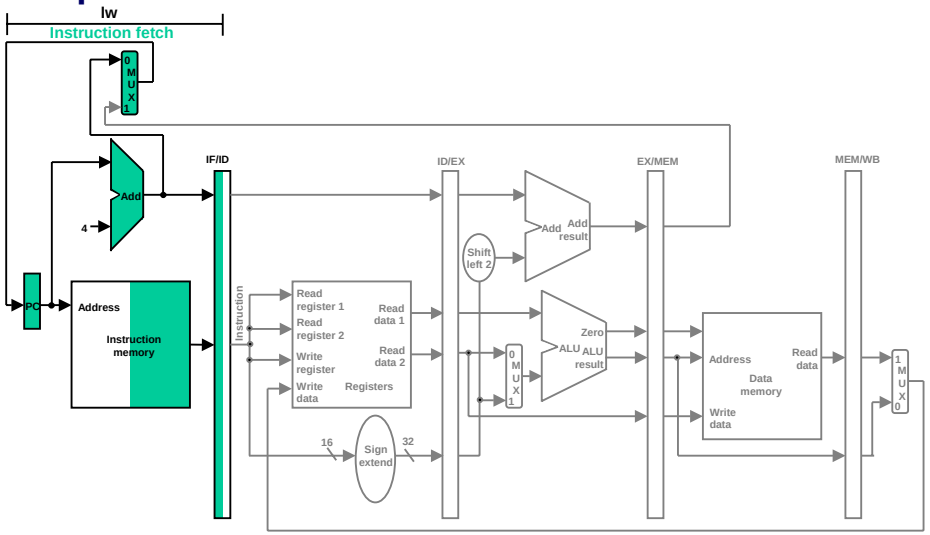


- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register.
- Similar arguments apply to every pipeline stage, so registers must be placed between all stages.

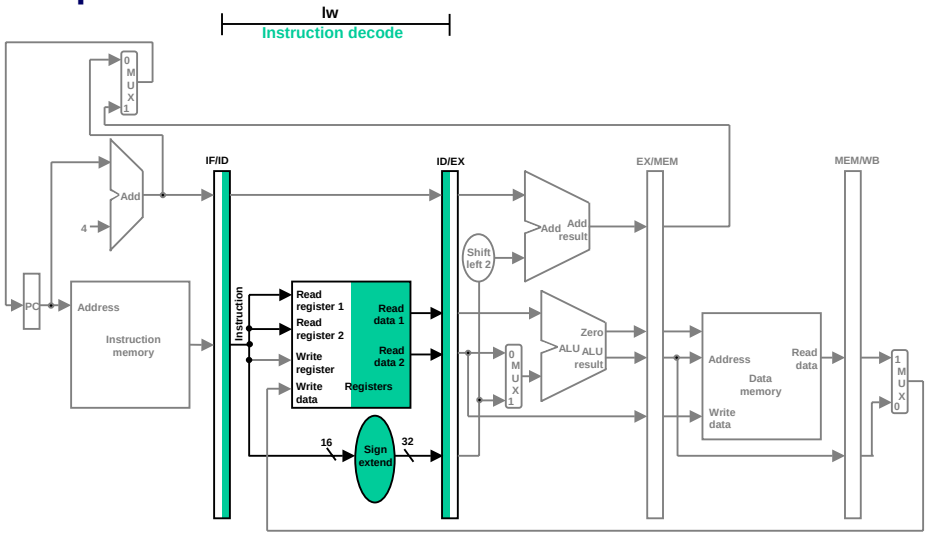
Notes on the Pipelined datapath

- The previous slide highlights the *pipeline registers* inserted between the pipeline stages.
- All instructions advance from one pipeline register to the next during each clock cycle.
- The registers are named for the two stages separated by that register; so the register separating the IF and ID stages is called the IF/ID pipeline register
- The write back stage does not need a register since the register file is updated at the end of that stage – a separate register would be redundant.

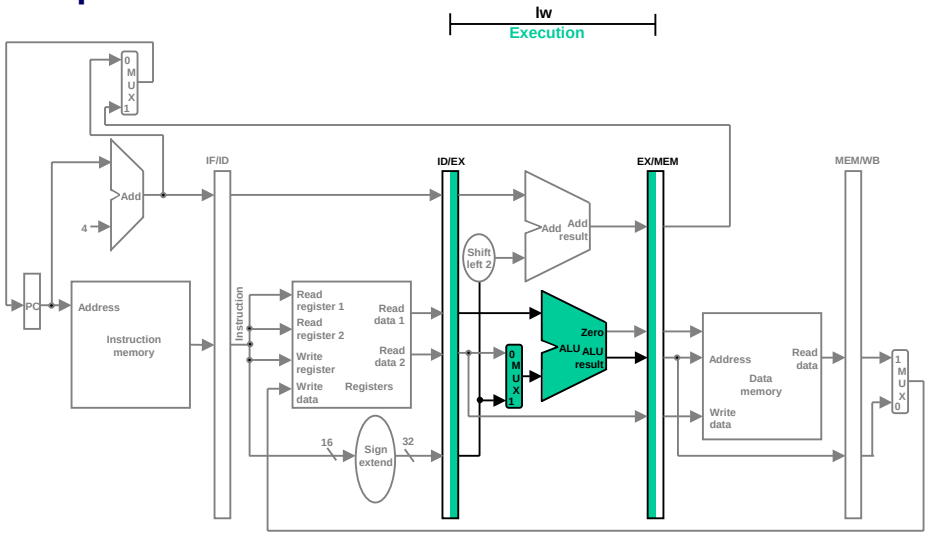
Pipelined LW execution: Instruction fetch



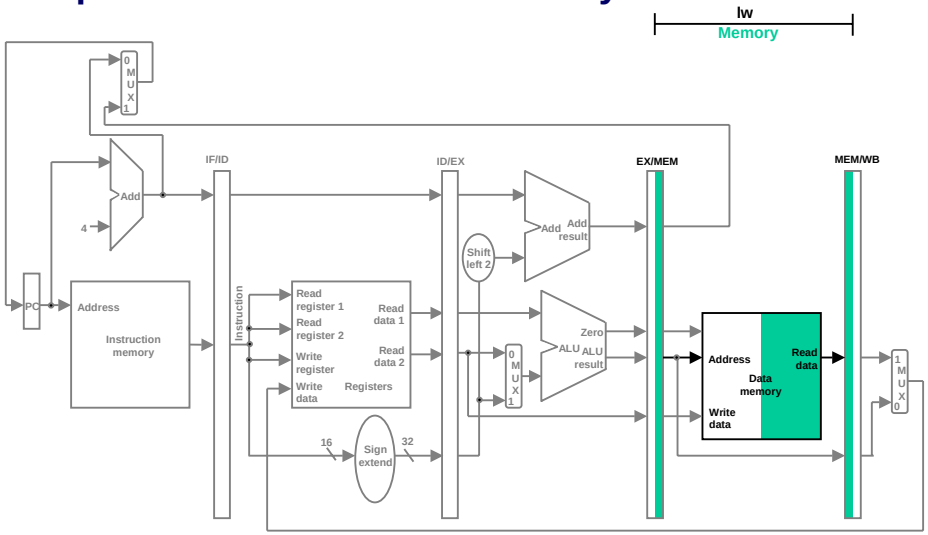
Pipelined LW execution: Instruction decode



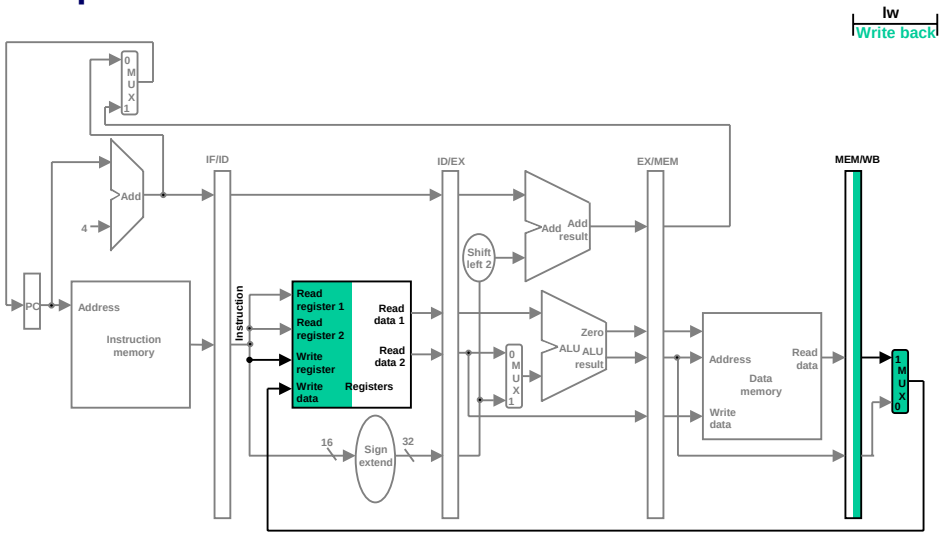
Pipelined LW execution: Execution



Pipelined LW execution: Memory access



Pipelined LW execution: Write back



Notes on pipelined LW execution

- The previous five slides show the active portions of the datapath as a load instruction progresses through the pipeline
- The *right half* of registers are highlighted as they are *read* and the *left half* is highlighted as they are being written
- In detail, the five stages are as follows:
  1. *Instruction fetch:* the instruction is read from memory using the address in the PC and is stored in IF/ID. PC is incremented by 4 and written back ready for the next instruction. The new PC value is stored in IF/ID for instructions such as beq. We store all data that may be needed by subsequent stages.
  2. *Instruction decode…* see over

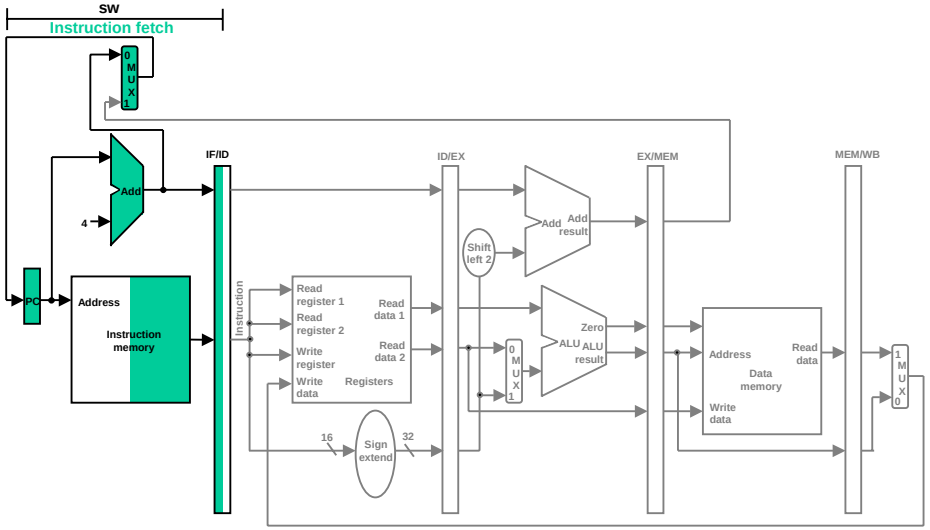
Notes on pipelined LW execution

- Pipeline stage details continued:
  2. *Instruction decode and register file read:* the immediate field is retrieved from IF/ID and sign extended; the two source registers are read and all three values are stored in ID/EX along with everything else that may be needed by the instruction during a later clock cycle.
  3. *Execute or address calculation:* the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register are added using the ALU and the result is placed in the EX/MEM pipeline register.
  4. *Memory access:* data memory is read using the address from EX/MEM; the read data is loaded into the MEM/WB pipeline register.
  5. *Write back:* the data is read from MEM/WB and written to the register file.

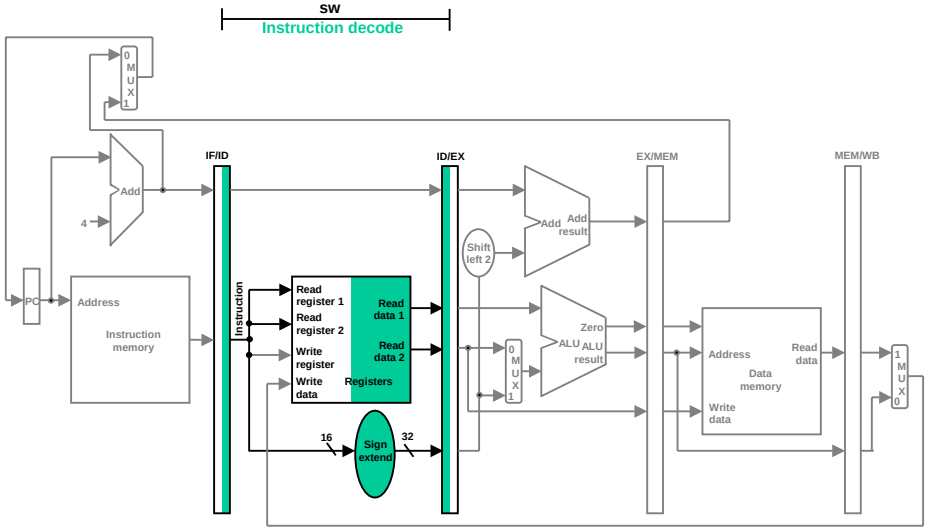
Other instructions?

- Let's just look at how a store instruction differs from a load instruction.

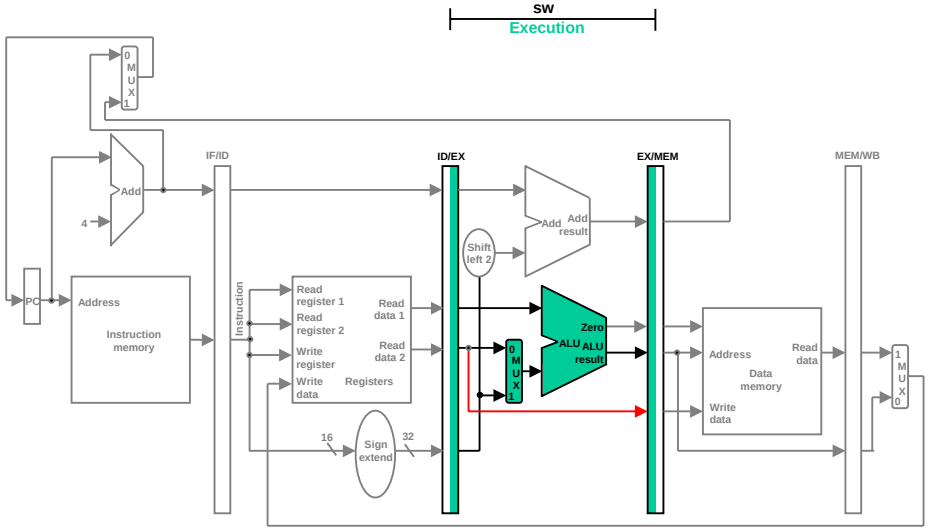
Pipelined SW execution: Instruction fetch



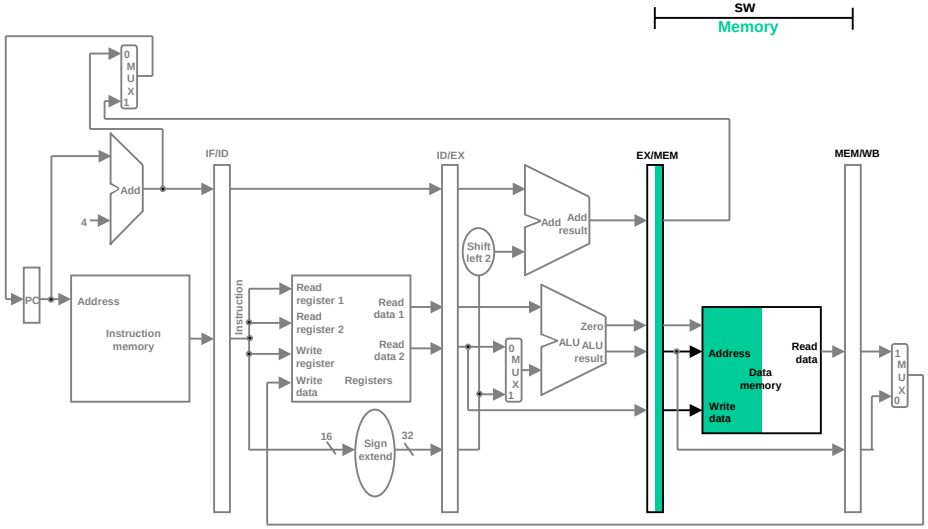
Pipelined SW execution: Instruction decode



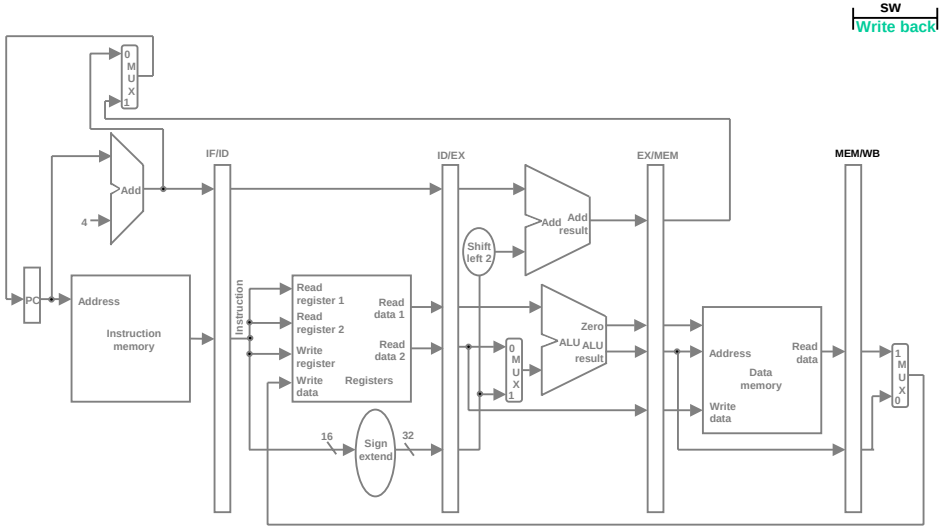
Pipelined SW execution: Execution



Pipelined SW execution: Memory access



Pipelined SW execution: Write back



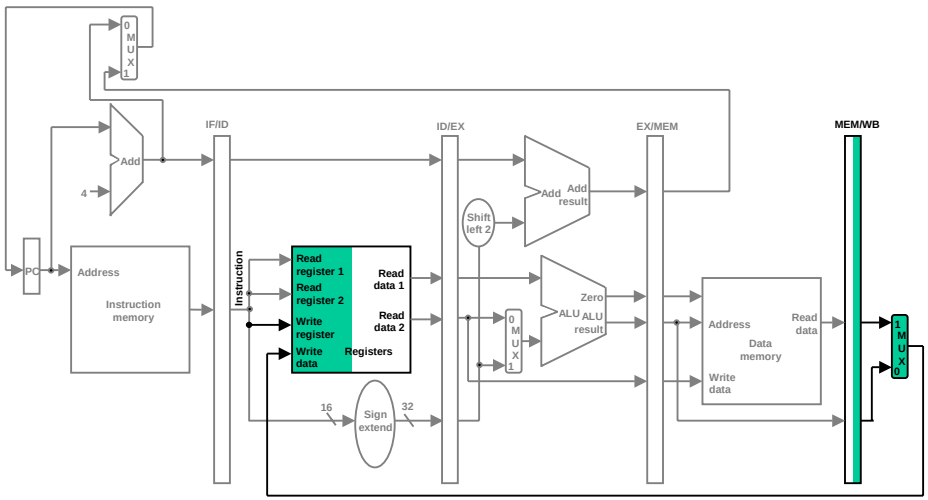
SW instruction execution

- **The first two stages are identical to those for the load instruction, but the latter stages become:**
  3. ***Execute and address calculation:*** **the effective address and the data to be stored are placed in the EX/MEM pipeline register**
  4. ***Memory access:*** **the data is written to memory. In order to make the data available during the MEM stage it had to be placed into the EX/MEM register during the EX stage.**
  5. ***Write back:*** **for this instruction, nothing happens in the write back stage. Since four instructions behind the store are already in progress, these instructions cannot be accelerated. An instruction therefore passes through a stage even if there is nothing to do since later instructions are already progressing at maximum rate.**

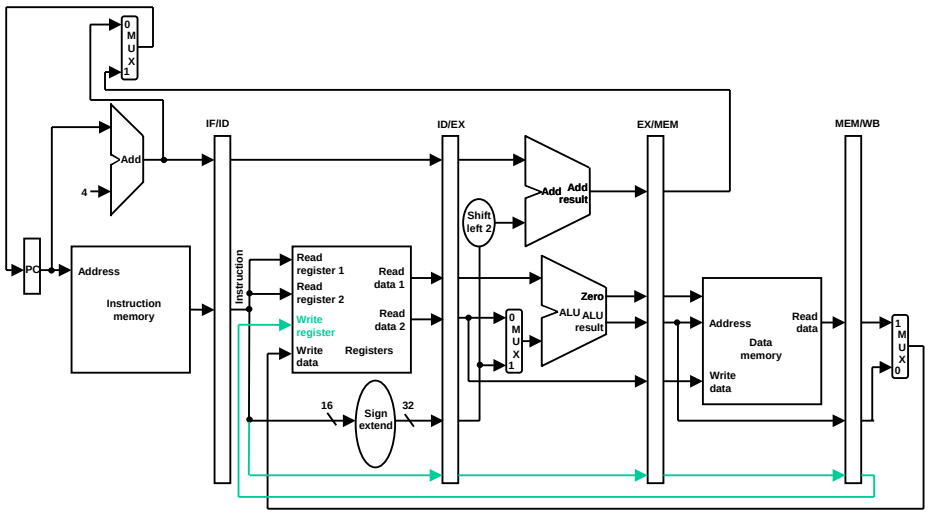
Key points

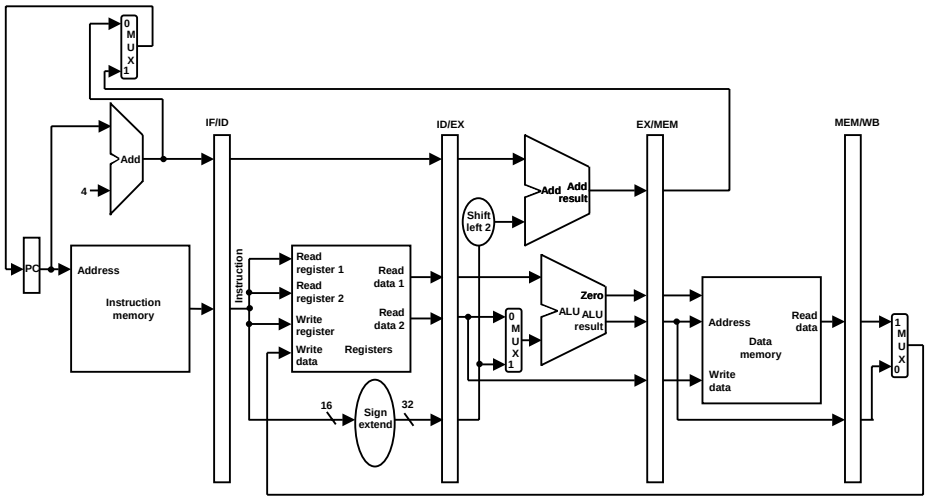
- **To pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise the information is lost when the next instruction enters that pipeline stage.**
- **Each logical component of the datapath – such as instruction memory, register read ports, ALU, data memory, and register write port – can be used only within a *single* pipeline stage; otherwise we would have a *structural hazard.* Hence these components, and their control, are associated with a single pipeline stage.**

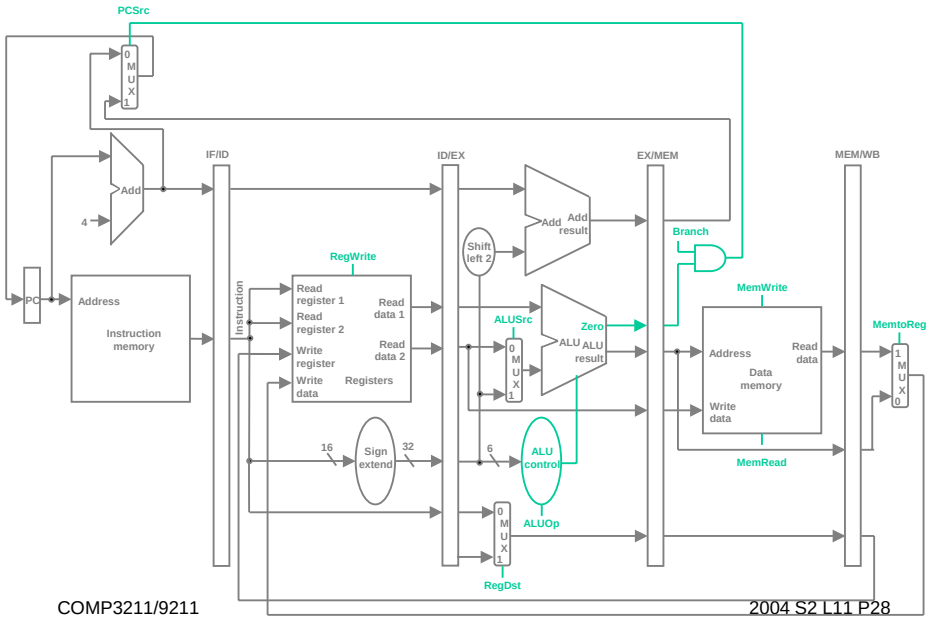
Bug in design of write back stage?



Corrected design: pipe the destination register number through the pipeline registers as well!



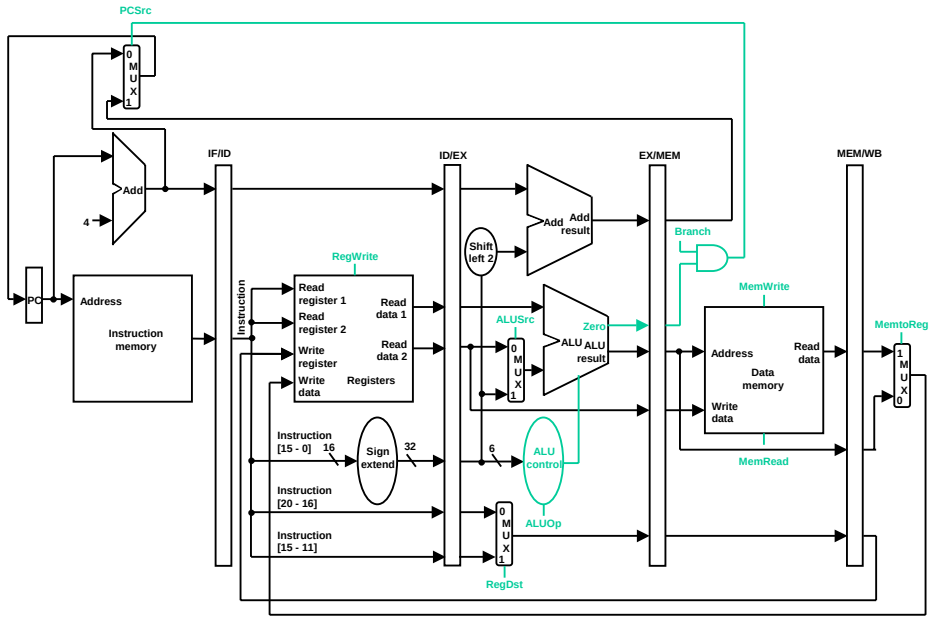




Pipelined control

- **Add control to pipelined datapath just like we added it to the simple datapath:**
  1. **Label control points**
  2. **Determine the control settings for each stage of execution of every instruction**
  3. **Design control logic to implement control**
     - **Rather than design a central controller, we implement small control functions that are local to each stage**
     - **To handle sequencing, control inputs are pipelined together with data and intermediate results**

Control points in the pipelined datapath



Notes:

- **To specify control for the pipeline, we need only set the control values during each pipeline stage.**
- **Since each control line is associated with a component that is active in only one pipeline stage, we divide the control lines into five groups according to the pipeline stage:**
  1. ***Instruction fetch:*** **IM read and PC write signals always asserted, so nothing to control.**
  2. ***Instruction decode/register file read:*** **The same function occurs every cycle, so no control is necessary.**
  3. ***Execution/address calculation:*** **RegDst, ALUOp, and ALUSrc need to be set.**
  4. ***Memory access:*** **Branch, MemRead, and MemWrite are set.**
  5. ***Write back:*** **MemtoReg and RegWrite need to be specified.**
- **Since the pipeline registers are written to each cycle, there is no need to control these.**

Recall ALU control settings

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Recall remaining control settings

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the write register comes from the rt field (bits 20 – 16)	The register destination number for the write register comes from the rd field (bits 15 – 11)
RegWrite	None	The register on the write register input is written with the value on the write data input
ALUSrc	The second ALU operand comes from the second register file output	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC+4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

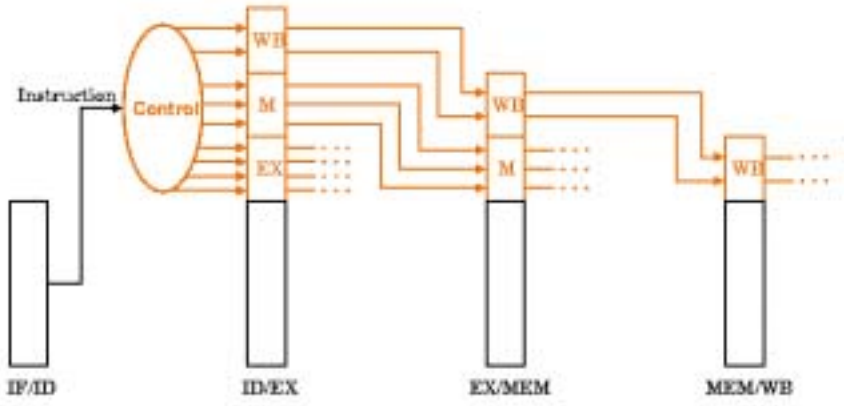
Control settings for selected MIPS instructions organised into pipeline stages

	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op2	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

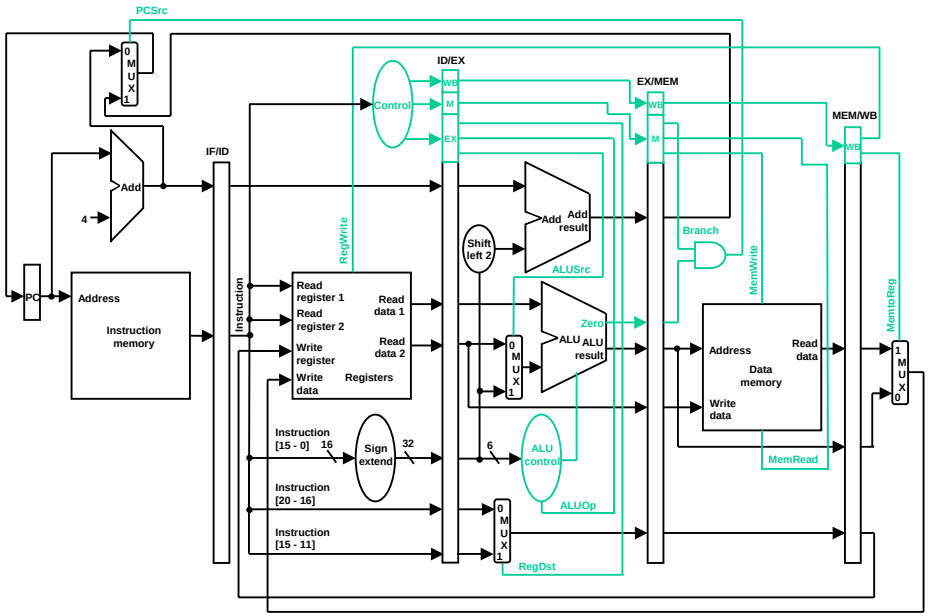
- **Implementing control means setting the control lines to these values in each stage for each instruction.**
- **The easiest way to do this is to extend the pipeline registers to include control information**

Pipelining the control values for the final 3 stages

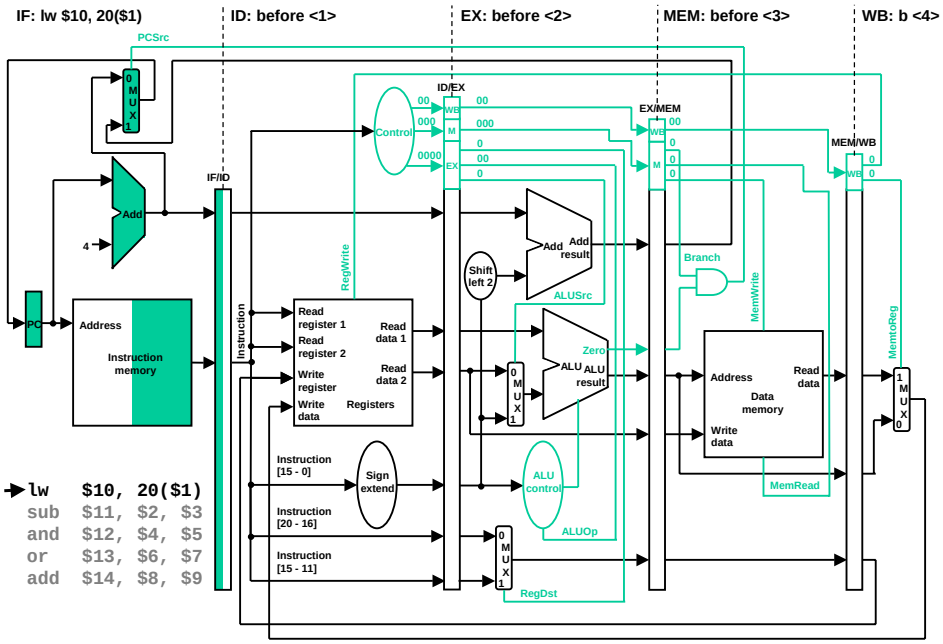
- **Since the control lines start with the EX stage, we can create the control information during the instruction decode stage**
- **These control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline**



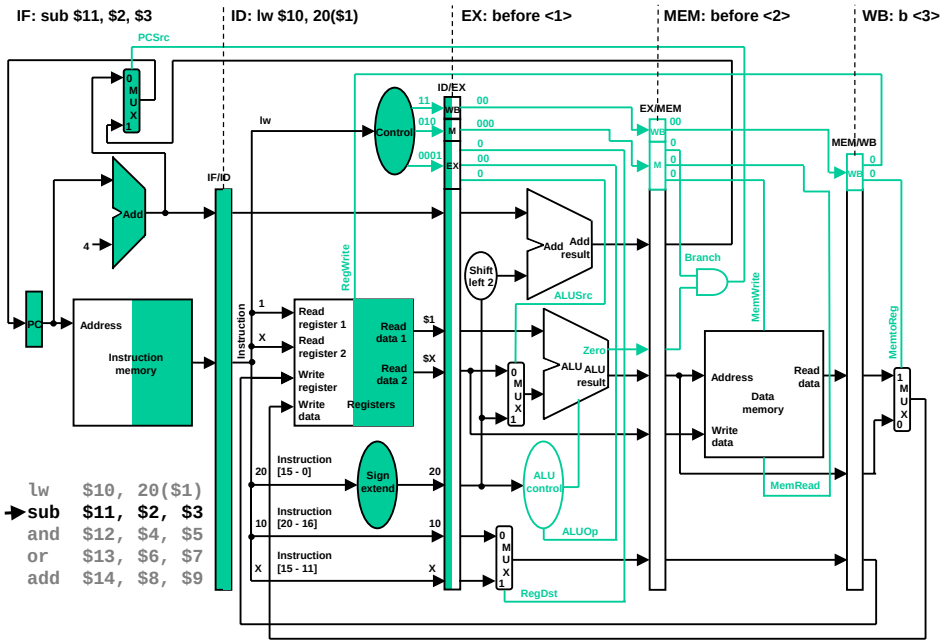
Pipelined datapath and control



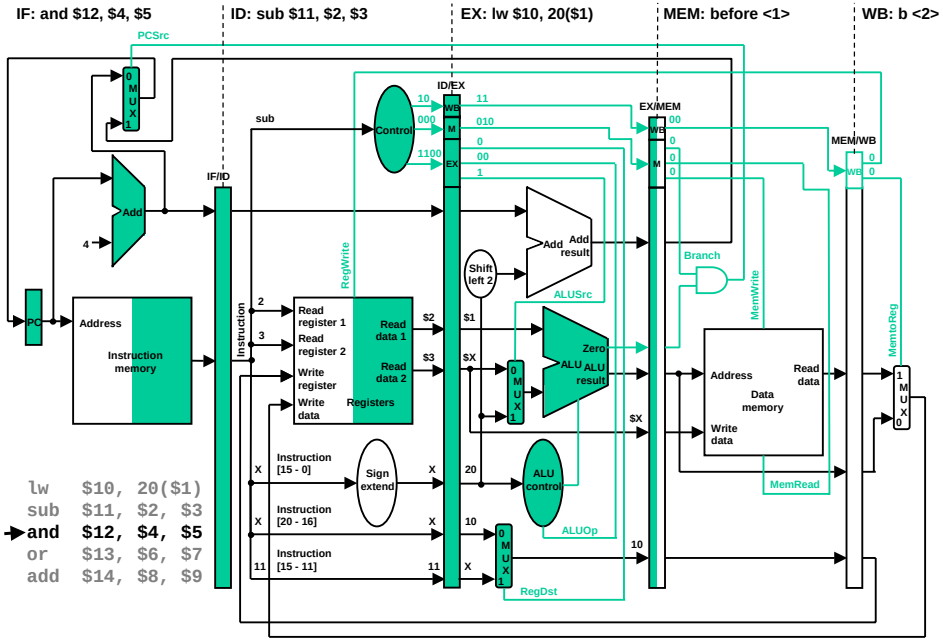
Pipelined program: clock cycle 1/9



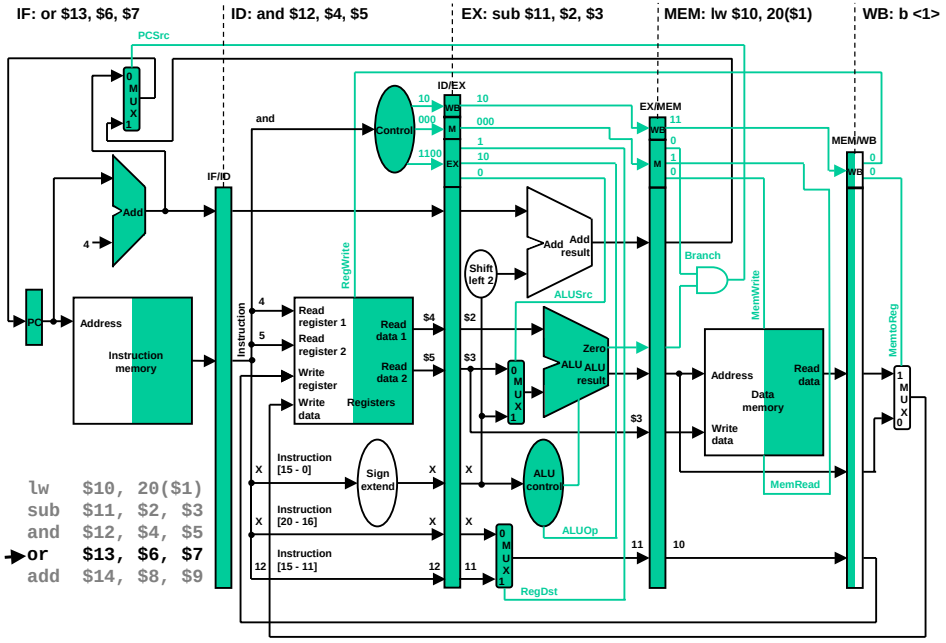
Pipelined program: clock cycle 2/9



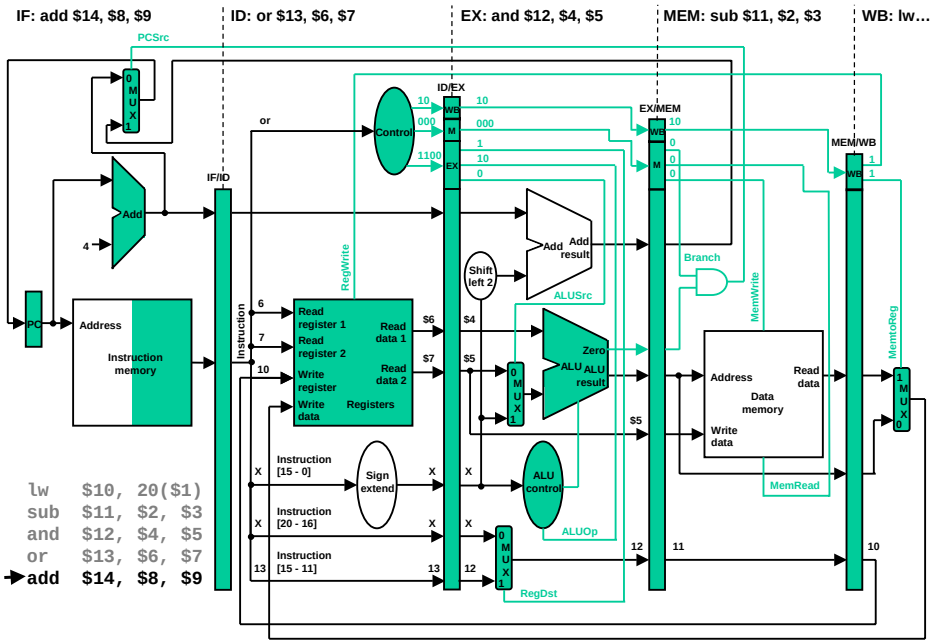
Pipelined program: clock cycle 3/9



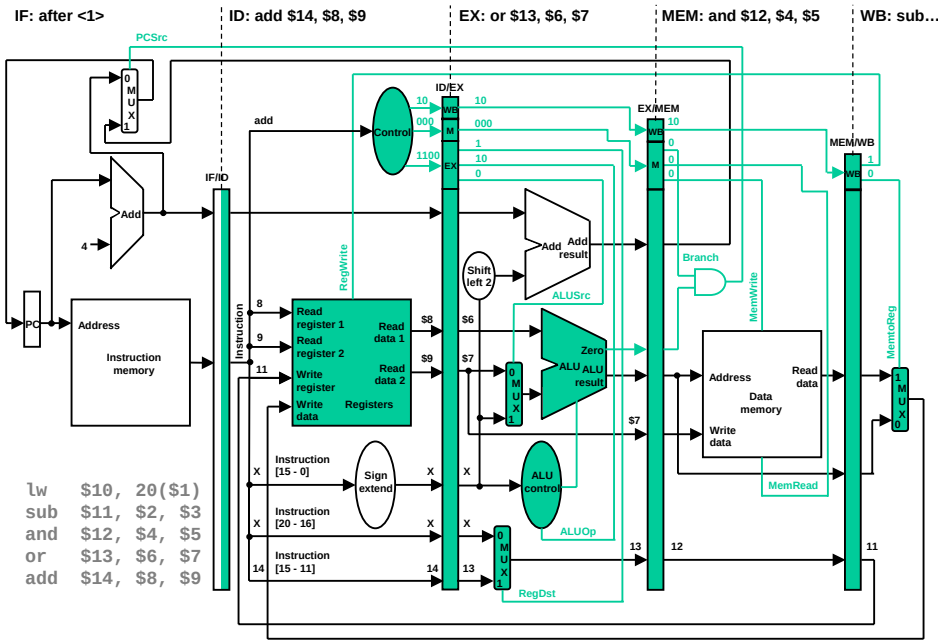
Pipelined program: clock cycle 4/9



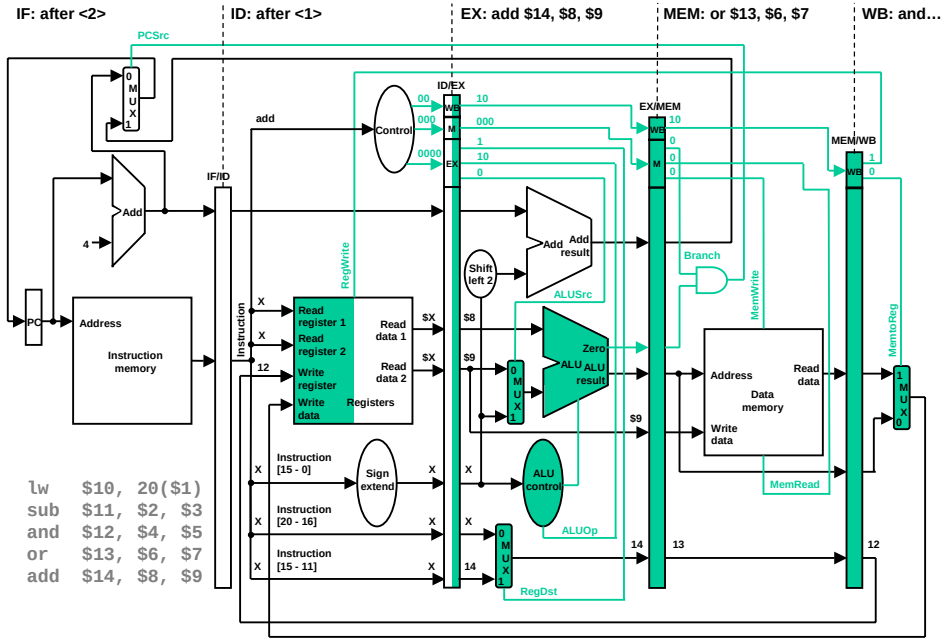
Pipelined program: clock cycle 5/9



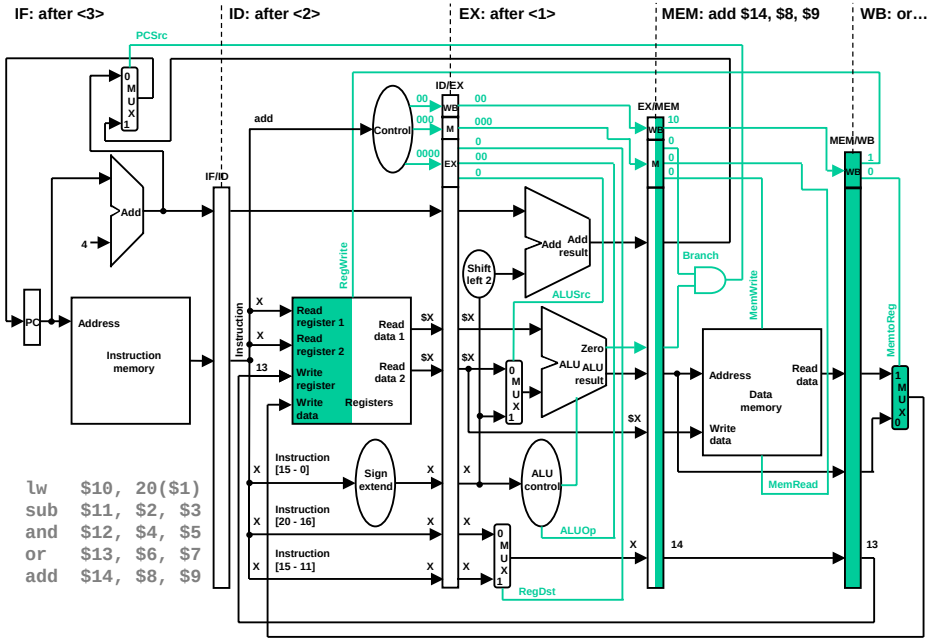
Pipelined program: clock cycle 6/9



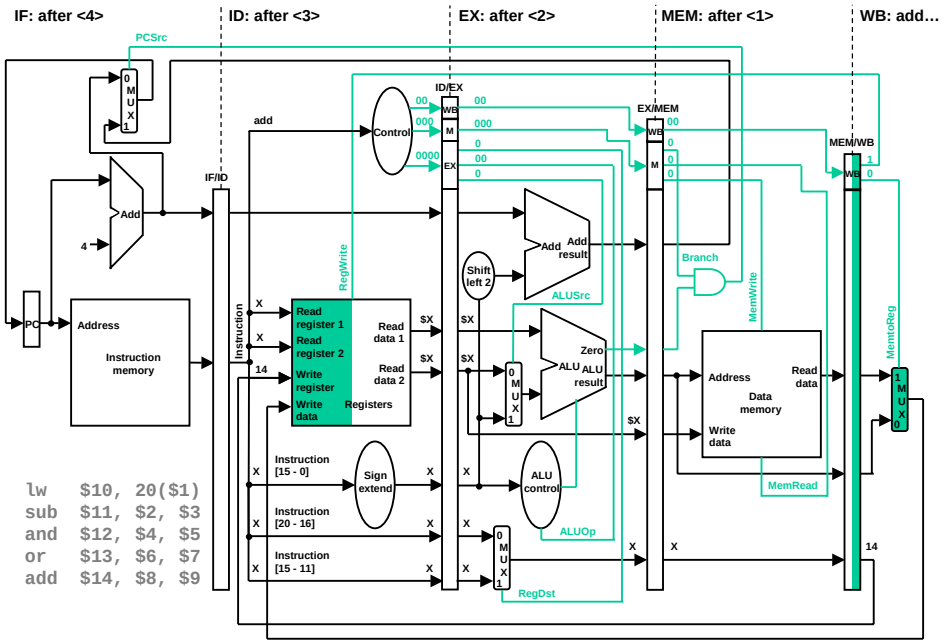
Pipelined program: clock cycle 7/9



Pipelined program: clock cycle 8/9



Pipelined program: clock cycle 9/9



Summary: Pipelining

- **What makes it easy**
  - **all instructions are the same length**
  - **just a few instruction formats**
  - **memory operands appear only in loads and stores**
- **What makes it hard?**
  - **structural hazards: suppose we had only one memory**
  - **control hazards: need to worry about branch instructions**
  - **data hazards: an instruction depends on a previous instruction**

Summary

- **Pipelining is a fundamental concept**
  - **multiple steps using distinct resources**
- **Utilize capabilities of the datapath by pipelined instruction processing**
  - **start next instruction while working on the current one**
  - **performance limited by length of longest stage (plus fill/flush)**
  - **detect and resolve hazards**