

Pipelining: Implementation (1)

Lecturer: Dr. Hui Annie Guo
huig@cse.unsw.edu.au
K17-501F (ext. 57136)

Overview

- Pipelined datapath structure
- Pipelined datapath operations
- Pipelined control

COMP3211/9211

2011S1 wk5_1 P2

Pipelining the datapath

- Recall the 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 that up to five instructions can be processed during a single clock cycle

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 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.

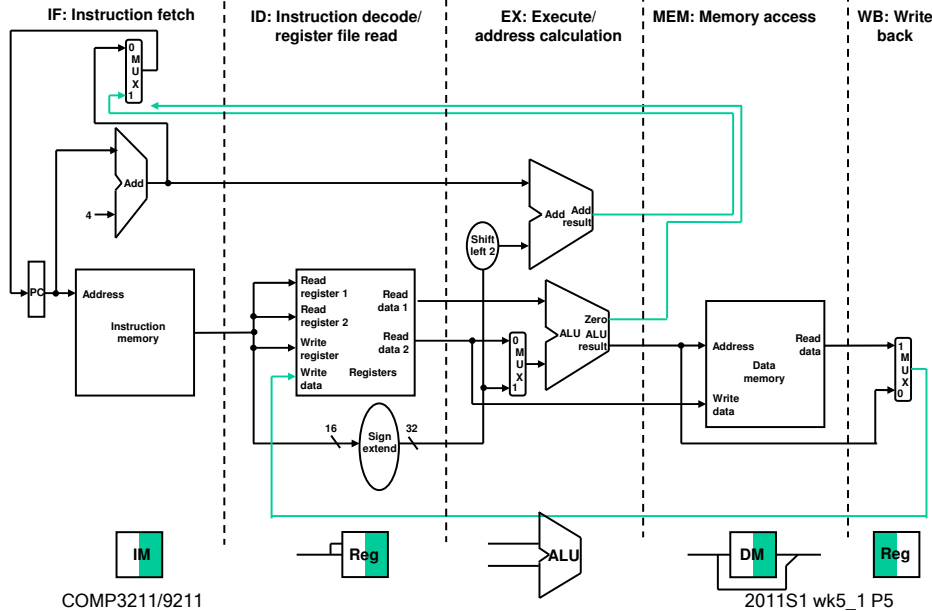
COMP3211/9211

2011S1 wk5_1 P3

COMP3211/9211

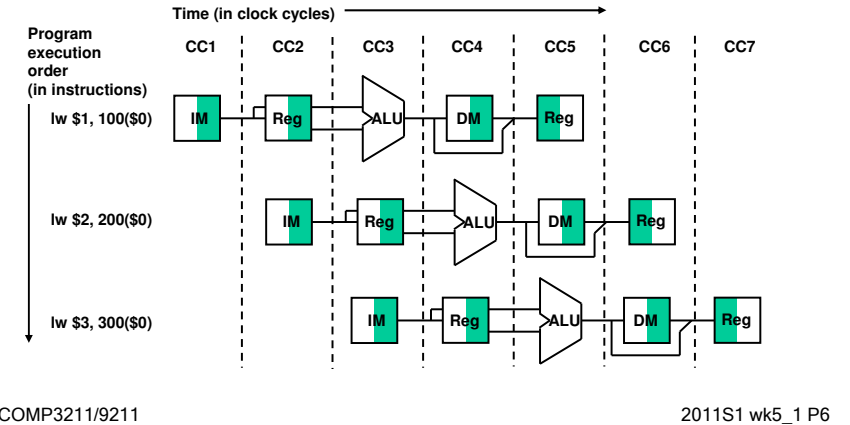
2011S1 wk5_1 P4

Single cycle datapath

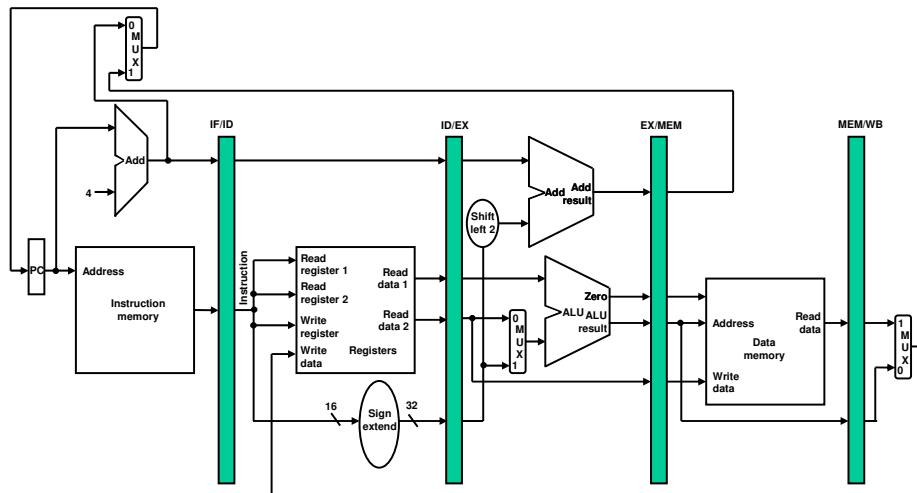


Towards pipelining...

- 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.



Pipelined datapath

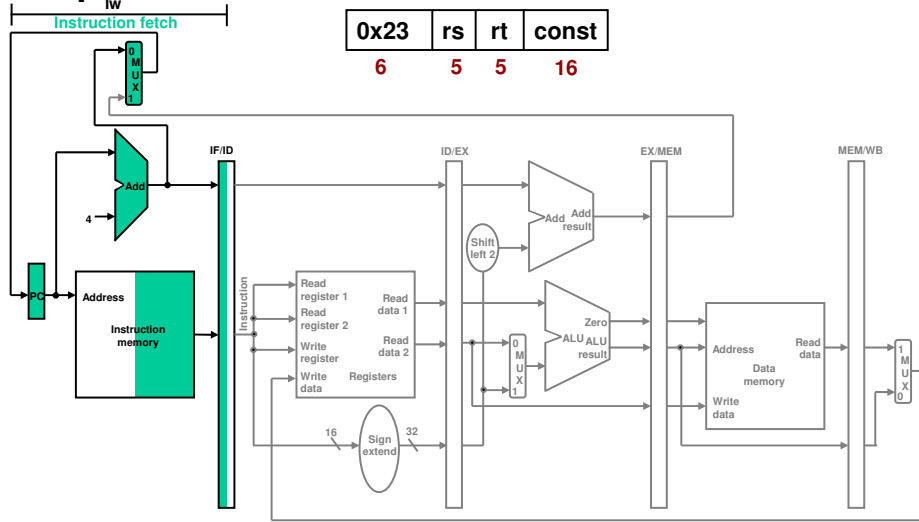


- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register.
- Similar arguments apply to every pipeline stage, so registers must be placed between all stages.

Notes on the pipelined datapath

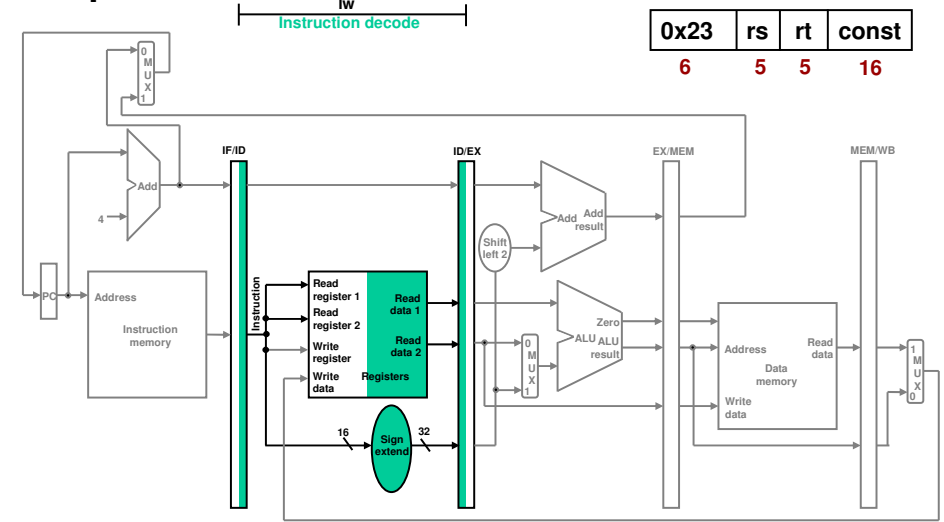
- The *registers* inserted between the pipeline stages are called *pipeline registers*.
- Each stage register is named after the two stages separated by that register
 - E.g. The register separating the IF and ID stages is called the *IF/ID pipeline register*
- All instructions advance from one pipeline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

Pipelined LW execution: Instruction fetch

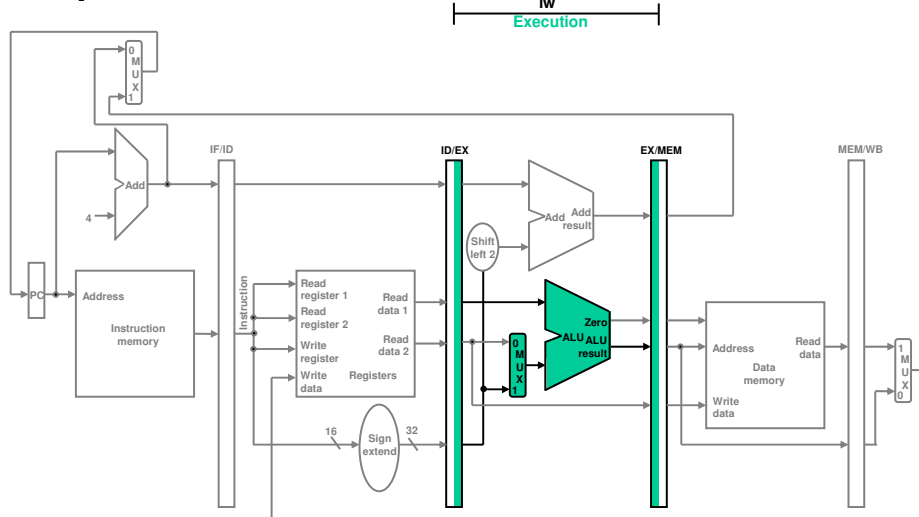


`lw rt, const(rs) # rt Mem(rs+const)`

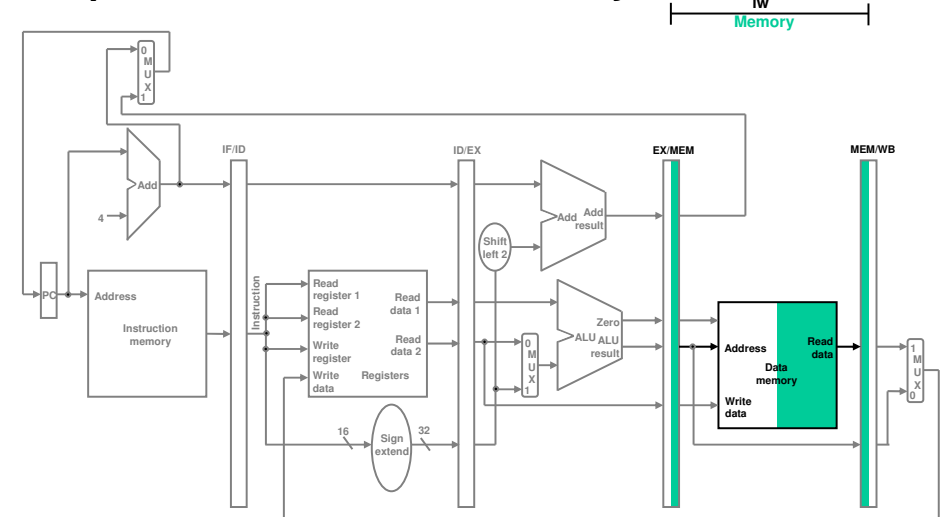
Pipelined LW execution: Instruction decode



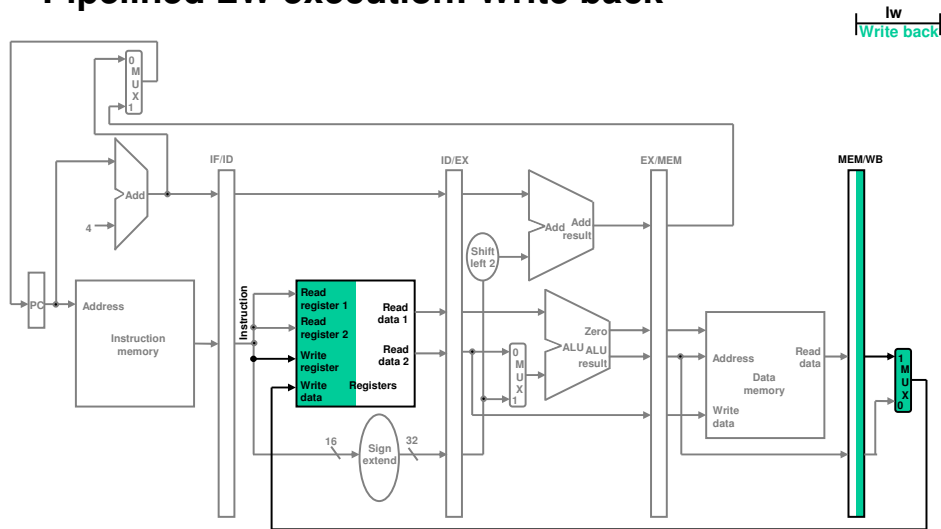
Pipelined LW execution: Execution



Pipelined LW execution: Memory access



Pipelined LW execution: Write back



COMP3211/9211

2011S1 wk5_1 P13

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 being *read* and the *left half* is highlighted as they are being *written*
- In detail, the five stages are working 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.

COMP3211/9211

2011S1 wk5_1 P14

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.

COMP3211/9211

2011S1 wk5_1 P15

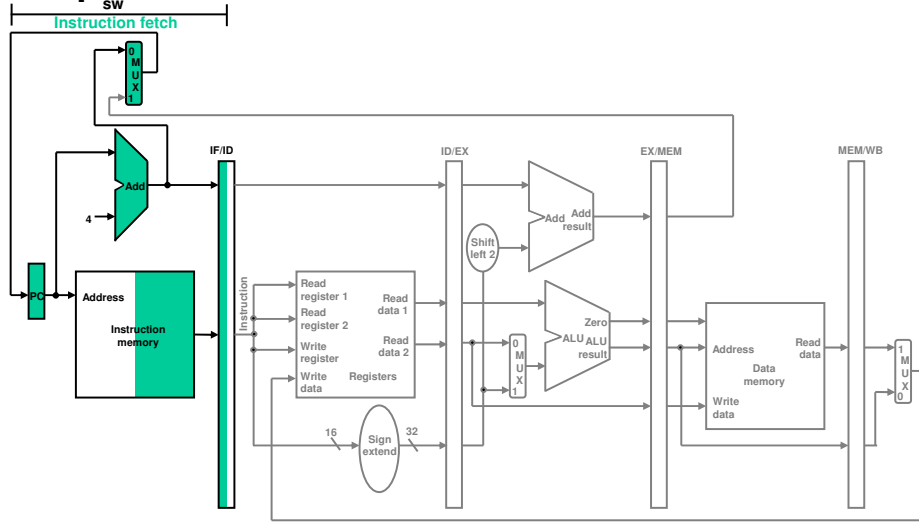
Other instructions?

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

COMP3211/9211

2011S1 wk5_1 P16

Pipelined SW execution: Instruction fetch

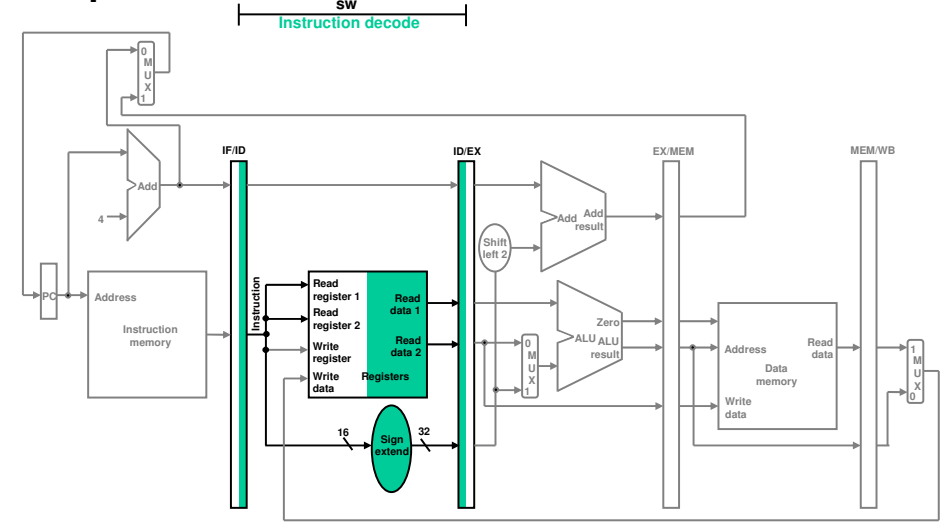


`sw rt, const(rs) # Mem(rs+const) rt`

COMP3211/9211

2011S1 wk5_1 P17

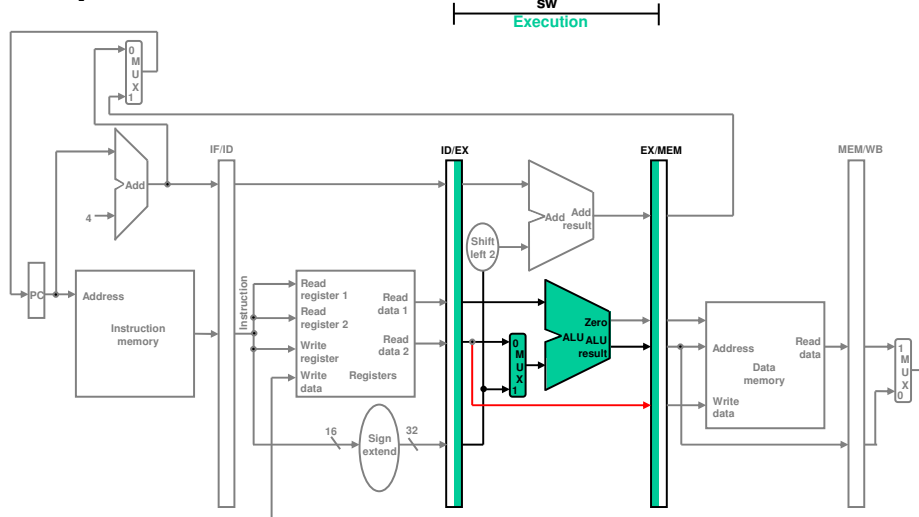
Pipelined SW execution: Instruction decode



COMP3211/9211

2011S1 wk5_1 P18

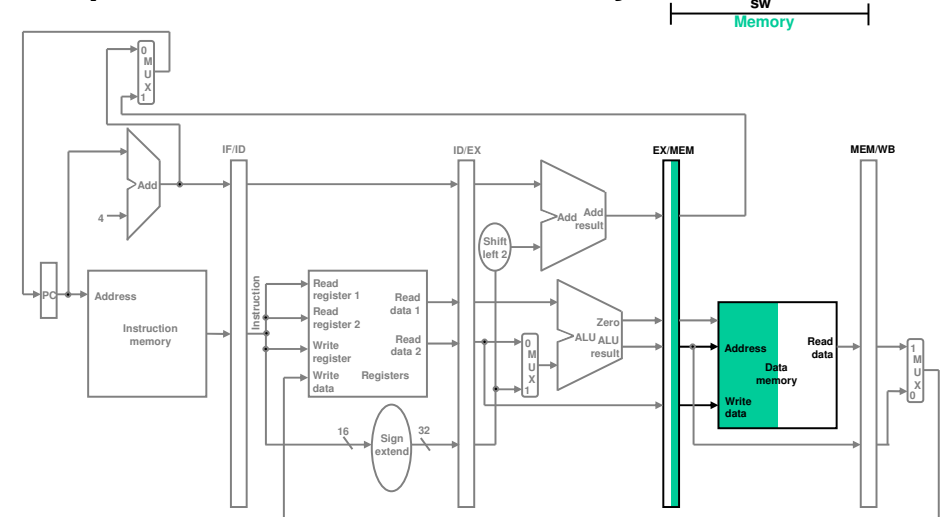
Pipelined SW execution: Execution



COMP3211/9211

2011S1 wk5_1 P19

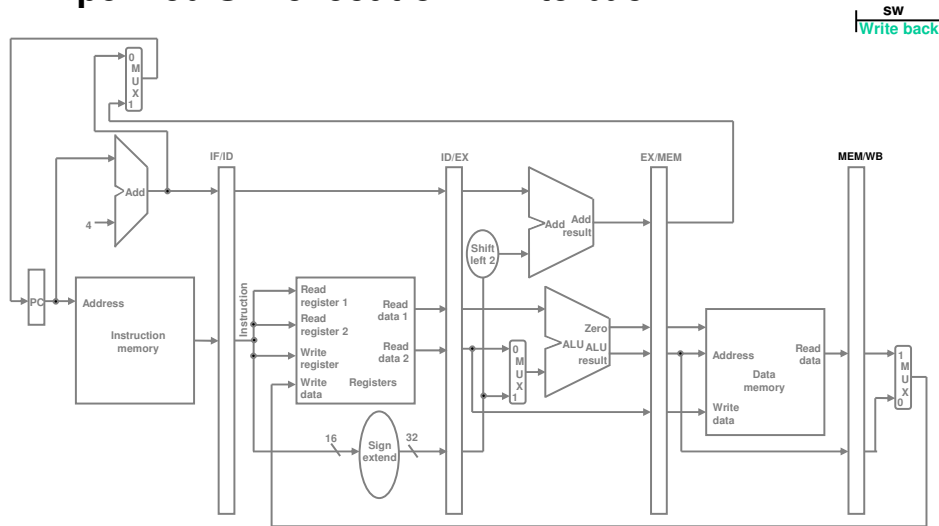
Pipelined SW execution: Memory access



COMP3211/9211

2011S1 wk5_1 P20

Pipelined SW execution: Write back



COMP3211/9211

2011S1 wk5_1 P21

SW instruction execution

- The first three stages are identical to those for the load instruction, but the latter stages become:
 - 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.
 - Write back:** for this instruction, nothing happens in the write back stage.

COMP3211/9211

2011S1 wk5_1 P22

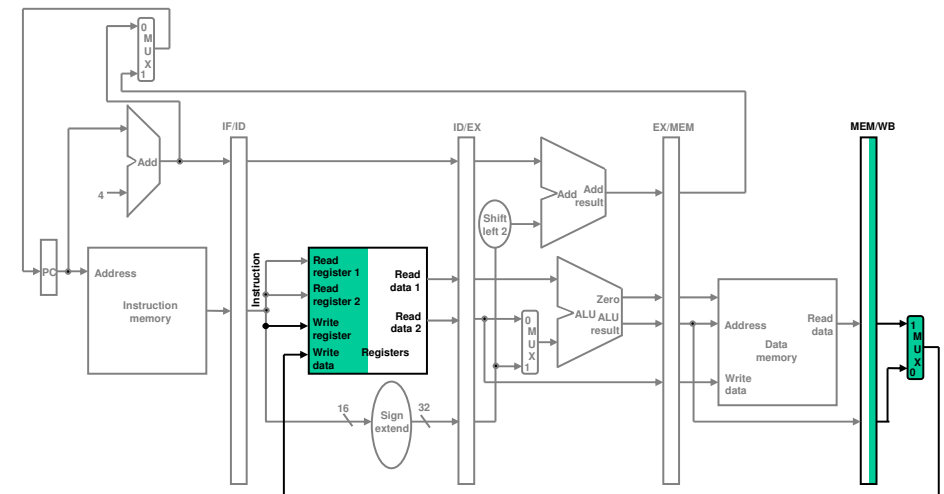
Key points

- 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.

COMP3211/9211

2011S1 wk5_1 P23

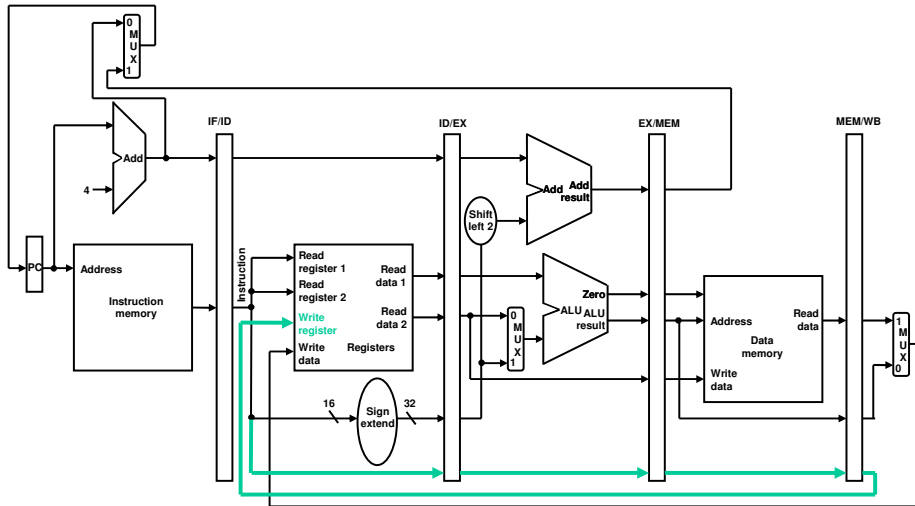
Bug in design of write back stage?



COMP3211/9211

2011S1 wk5_1 P24

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



COMP3211/9211

2011S1 wk5_1 P25

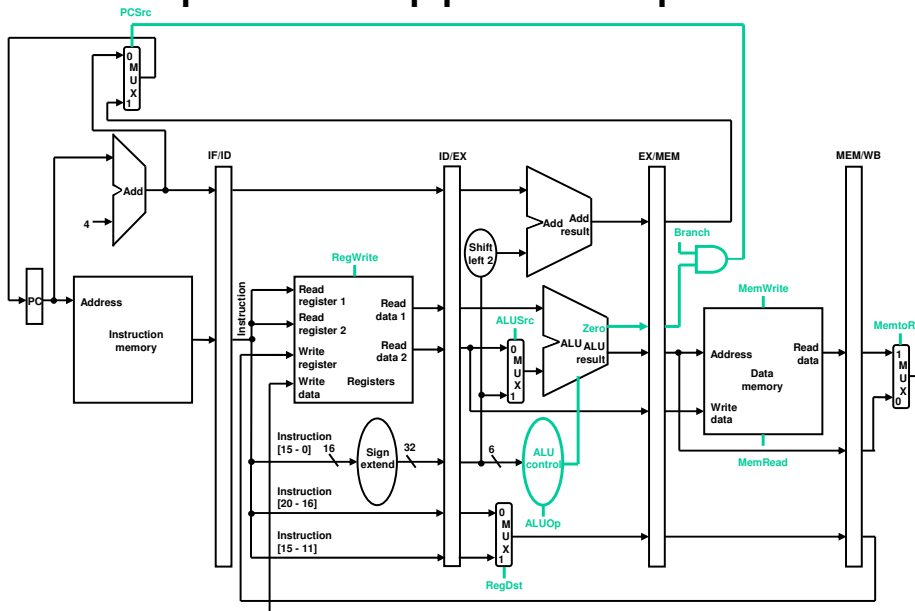
Pipelined control

- Add control to pipelined datapath just like we added it to the single cycle datapath:
 1. Label control points
 2. Determine the control settings for each stage of execution of every instruction
 3. Design control logic to implement the control
 - To handle sequencing, control inputs are pipelined together with data and intermediate results

COMP3211/9211

2011S1 wk5_1 P26

Control points in the pipelined datapath



Notes:

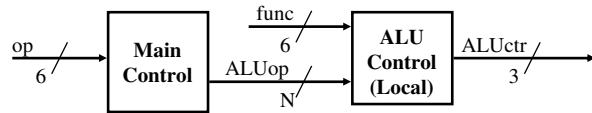
- To specify control for the pipeline, we need only set the control values during each pipeline stage.
- Since each control line is associated with a component that is active in only one pipeline stage, we divide the control lines into five groups according to the pipeline stage:
 1. *Instruction fetch*: 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.

COMP3211/9211

2011S1 wk5_1 P28

Recall: Control unit of single cycle processor

- We can use the same design for pipelined datapath

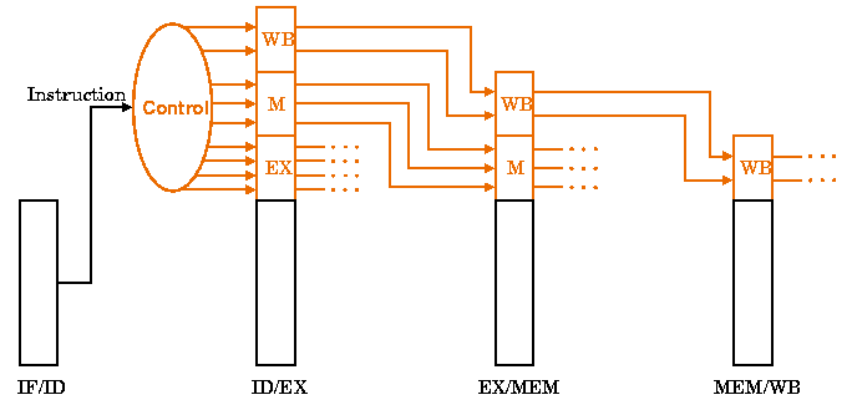


COMP3211/9211

2011S1 wk5_1 P29

Pipelining the control values for the final 3 stages

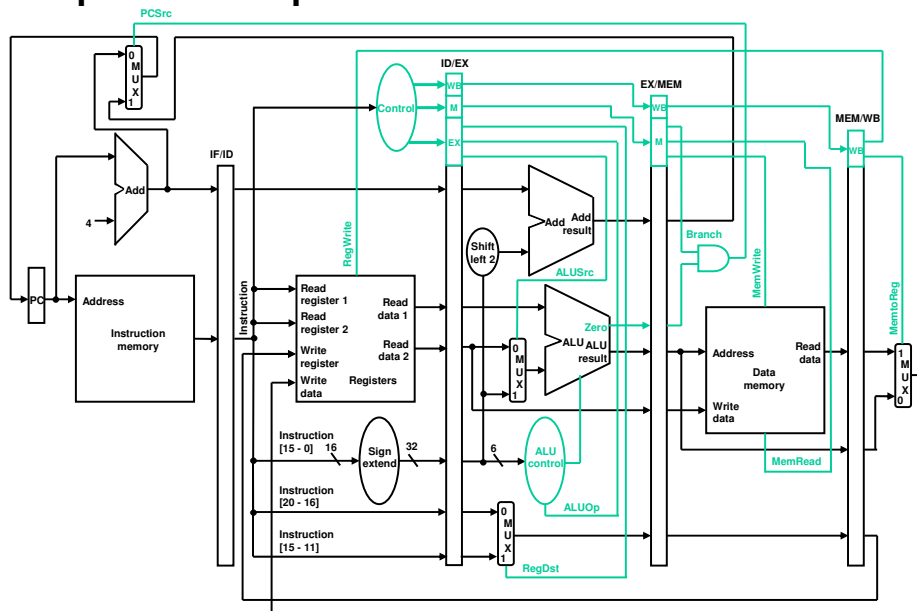
- Since the control lines start with the EX stage, we can create the control information during the instruction decode stage
- These control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline



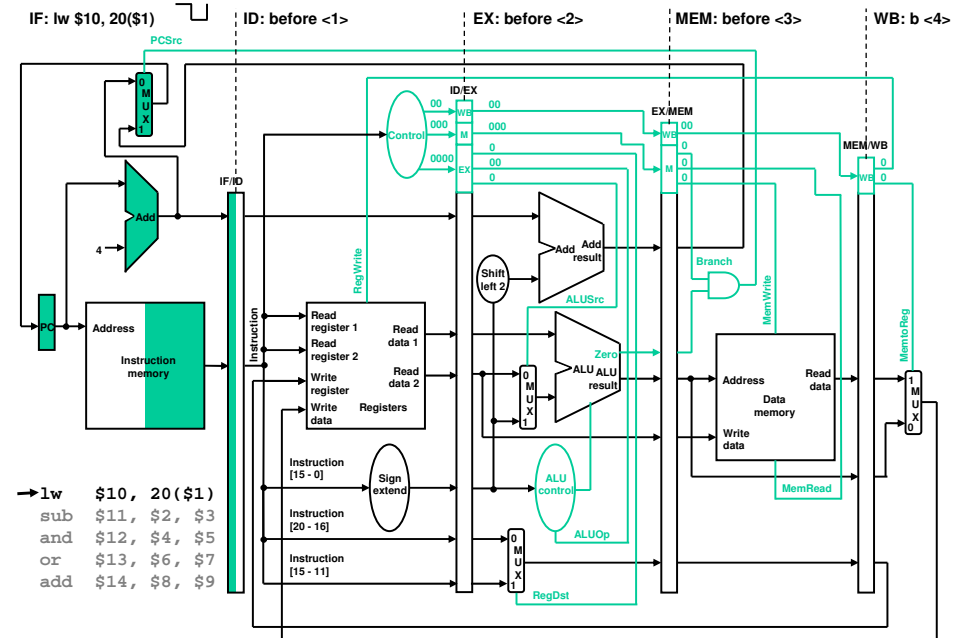
COMP3211/9211

2011S1 wk5_1 P30

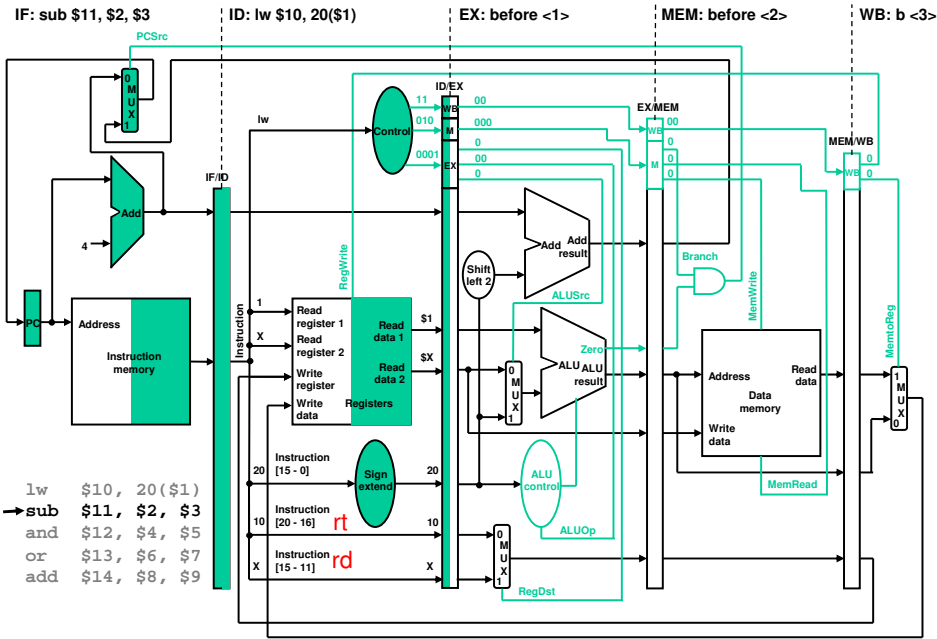
Pipelined datapath and control



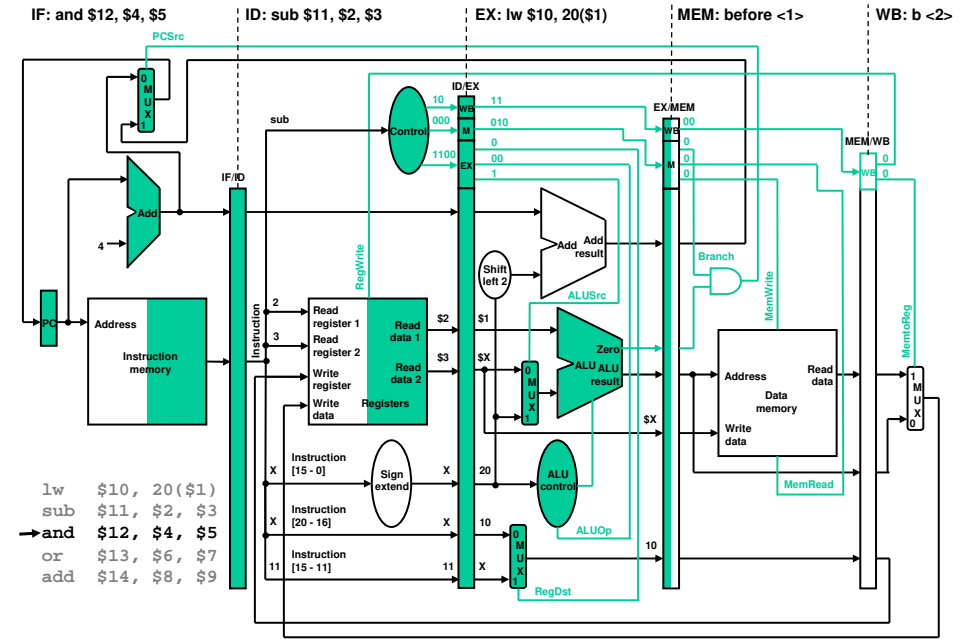
Pipelined program: clock cycle 1/9



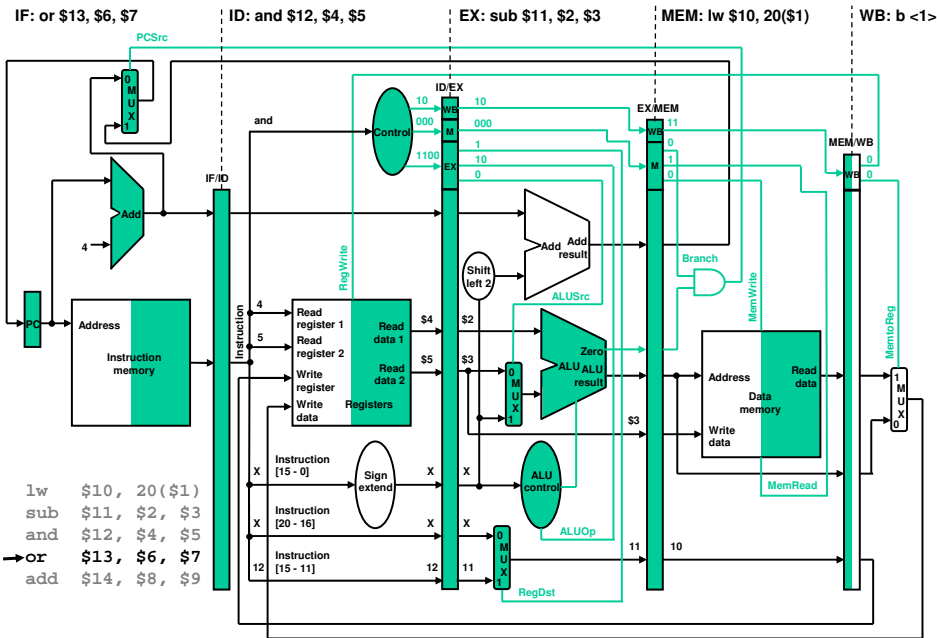
Pipelined program: clock cycle 2/9



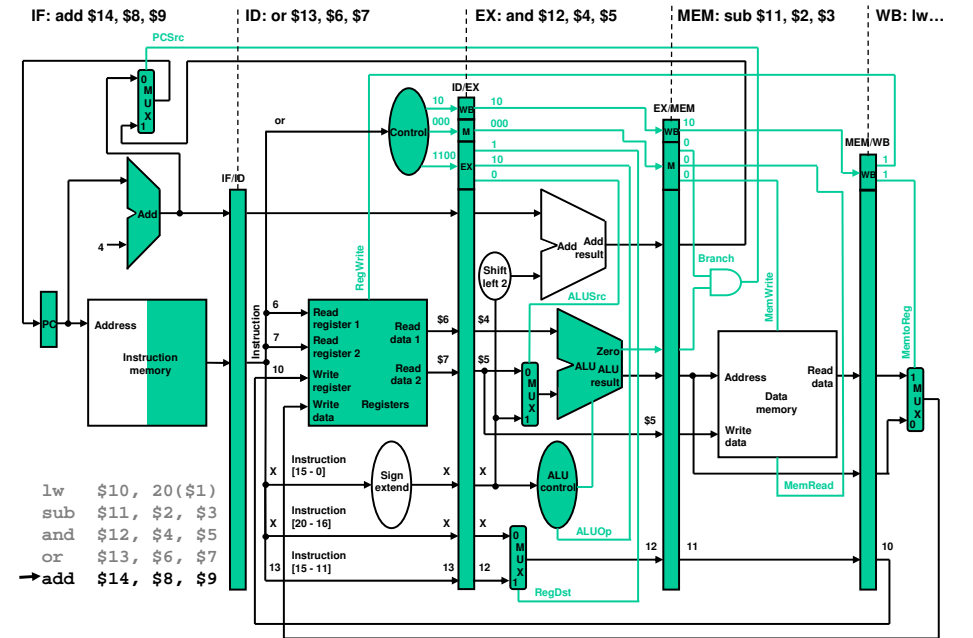
Pipelined program: clock cycle 3/9



Pipelined program: clock cycle 4/9



Pipelined program: clock cycle 5/9



Summary: Pipelining

- **Pipelining is a fundamental concept**
 - start next instruction while working on the current one
 - multiple steps using distinct resources
 - performance limited by the length of longest stage (plus fill/flush)
- **What makes it easy**
 - all instructions are of the same length
 - just a few instruction formats
 - Data memory accesses 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