Lecture 13 (Mon 10/27/2008)

- Lab #2 Software – Due Fri Oct 31 at 5pm
- Lab #2 Hardware – Due Fri Nov 7 at 5pm

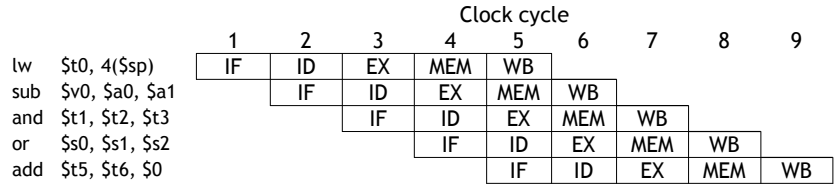
- Today's objectives:
  - Pipelining Datapath and control

Pipelined datapath and control

- Now we'll see a basic implementation of a pipelined processor.
  - The datapath and control unit share similarities with both the single-cycle and multicycle implementations that we already saw.
  - An example execution highlights important pipelining concepts.
- In future lectures, we'll discuss several complications of pipelining that we're hiding from you for now.

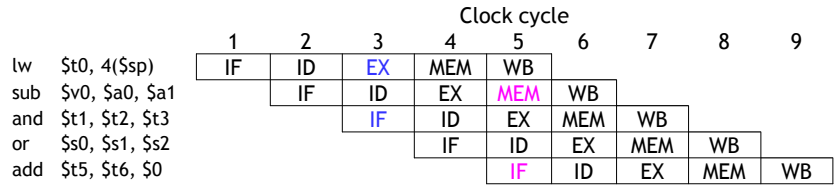
Pipelining concepts

- A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.
- This increases throughput, so programs can run faster.
  - One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.



Pipelined Datapath

- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.



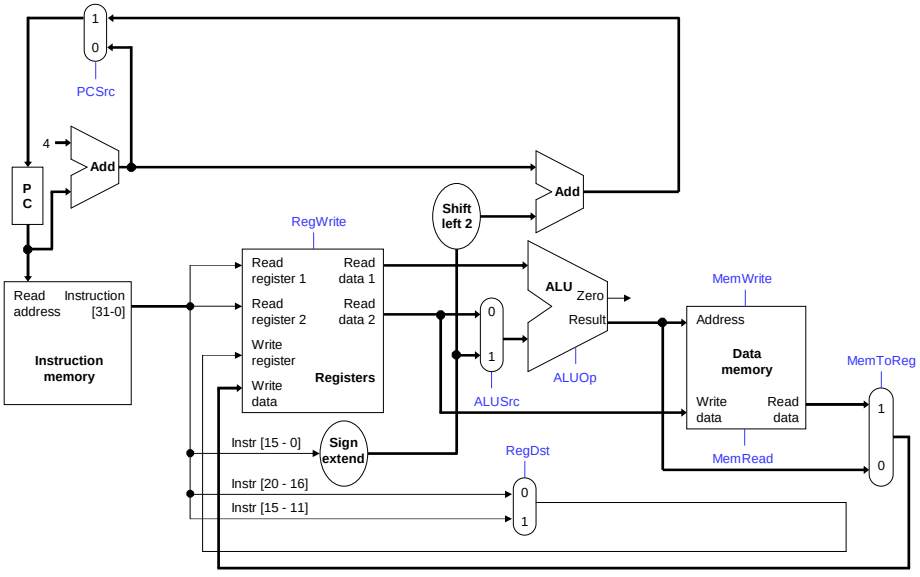
- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

One register file is enough

- We need only one register file to support both the ID and WB stages.

- Reads and writes go to separate ports on the register file.
- Writes occur in the first half of the cycle, reads occur in the second half.

Single-cycle datapath, slightly rearranged



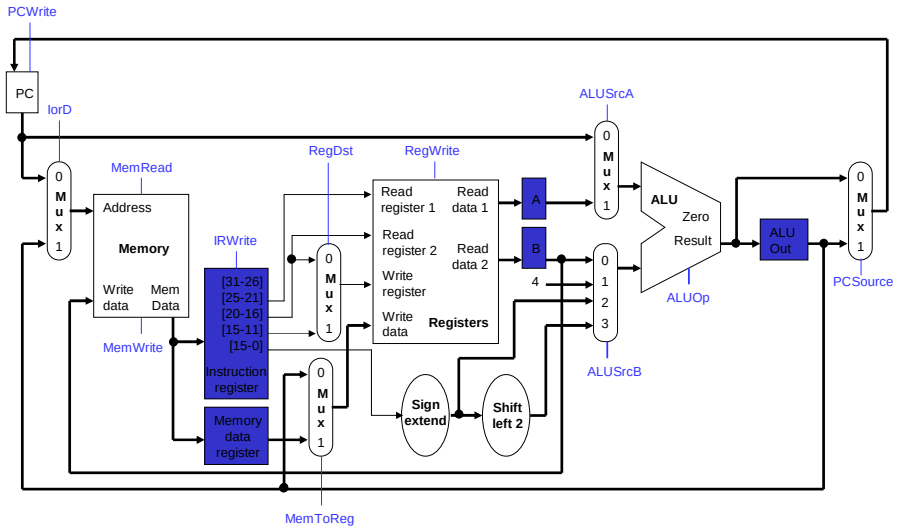
What's been changed?

- Almost nothing! This is equivalent to the original single-cycle datapath.
  - There are separate memories for instructions and data.
  - There are two adders for PC-based computations and one ALU.
  - The control signals are the same.
- Only some cosmetic changes were made to make the diagram smaller.
  - A few labels are missing, and the muxes are smaller.
  - The data memory has only one Address input. The actual memory operation can be determined from the MemRead and MemWrite control signals.
- The datapath components have also been moved around in preparation for adding pipeline registers.

Multiple cycles

- In pipelining, we also divide instruction execution into multiple cycles.
- Information computed during one cycle may be needed in a later cycle.
  - The instruction read in the IF stage determines which registers are fetched in the ID stage, what constant is used for the EX stage, and what the destination register is for WB.
  - The registers read in ID are used in the EX and/or MEM stages.
  - The ALU output produced in the EX stage is an effective address for the MEM stage or a result for the WB stage.
- We added several intermediate registers to the multicycle datapath to preserve information between stages, as highlighted on the next slide.

Registers added to the multi-cycle



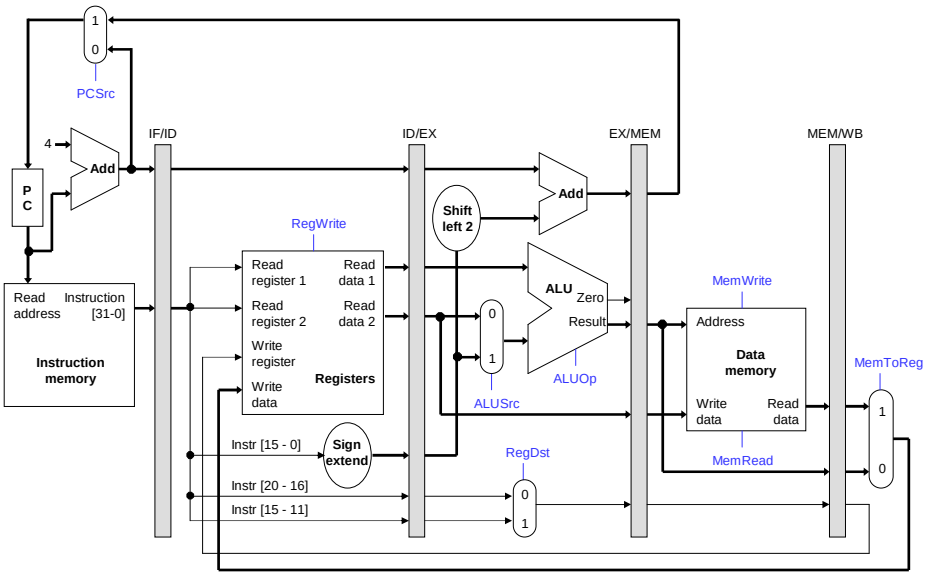
Pipeline registers

- We'll add intermediate registers to our pipelined datapath too.
- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
- The registers are named for the stages they connect.

IF/ID ID/EX EX/MEM MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done.

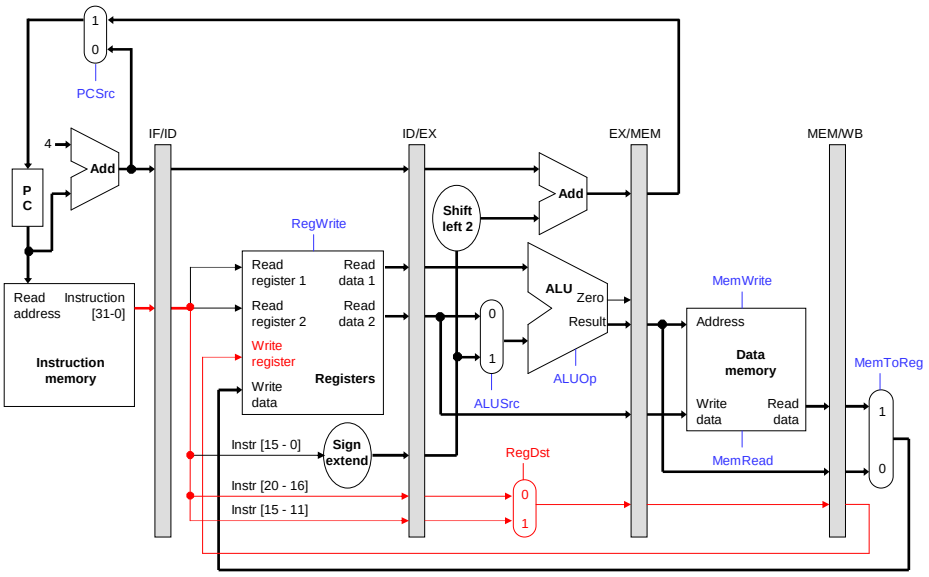
Pipelined datapath



Propagating values forward

- Any data values required in later stages must be propagated through the pipeline registers.
- The most extreme example is the destination register.
  - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn't updated until the *fifth* stage (WB).
  - Thus, the rd field must be passed through all of the pipeline stages, as shown in red on the next slide.
- Notice that we can't keep a single "instruction register" like we did before in the multicycle datapath, because the pipelined machine needs to fetch a new instruction every clock cycle.

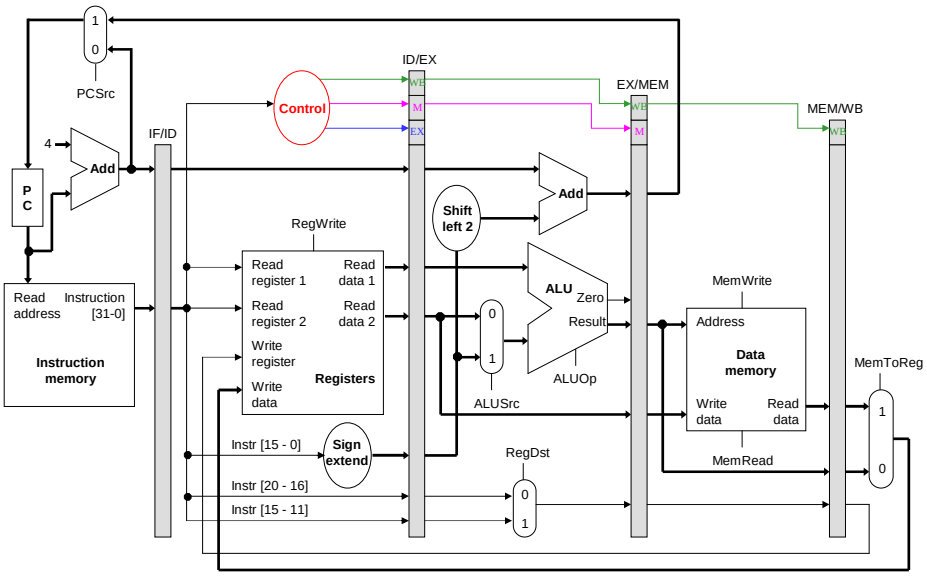
The destination register



What about control signals?

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.
- Control signals can be categorized by the pipeline stage that uses them.

Pipelined datapath and control

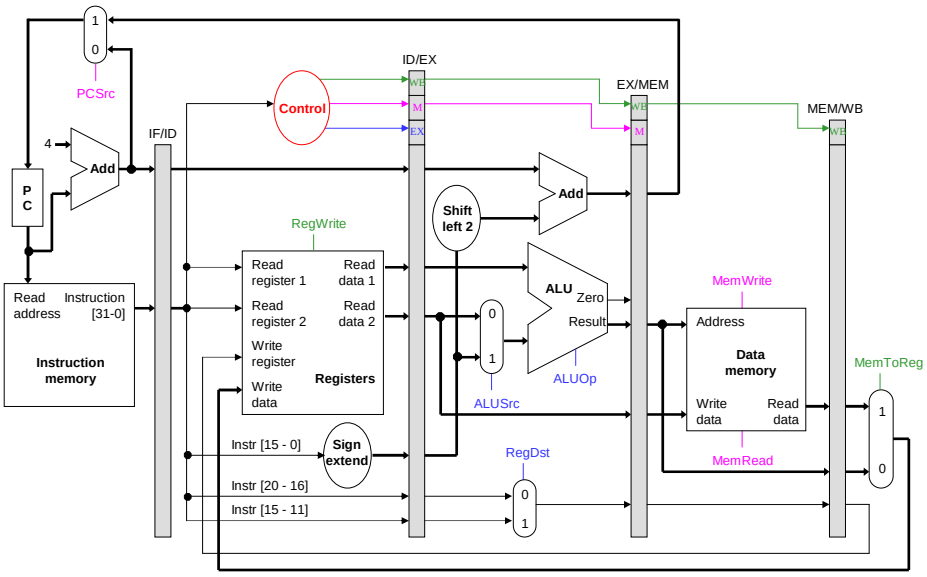


What about control signals?

- The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other data.
- Control signals can be categorized by the pipeline stage that uses them.

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

Pipelined datapath and control



Notes about the diagram

- The control signals are grouped together in the pipeline registers, just to make the diagram a little clearer.
- Not all of the registers have a write enable signal.
  - Because the datapath fetches one instruction per cycle, the PC must also be updated on each clock cycle. Including a write enable for the PC would be redundant.
  - Similarly, the pipeline registers are also written on every cycle, so no explicit write signals are needed.

An example execution sequence

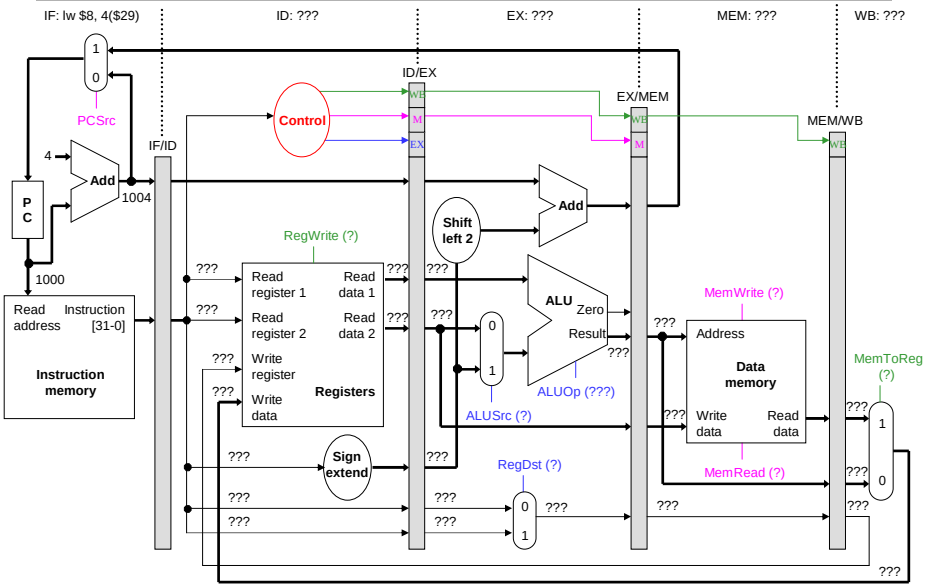
- Here's a sample sequence of instructions to execute.

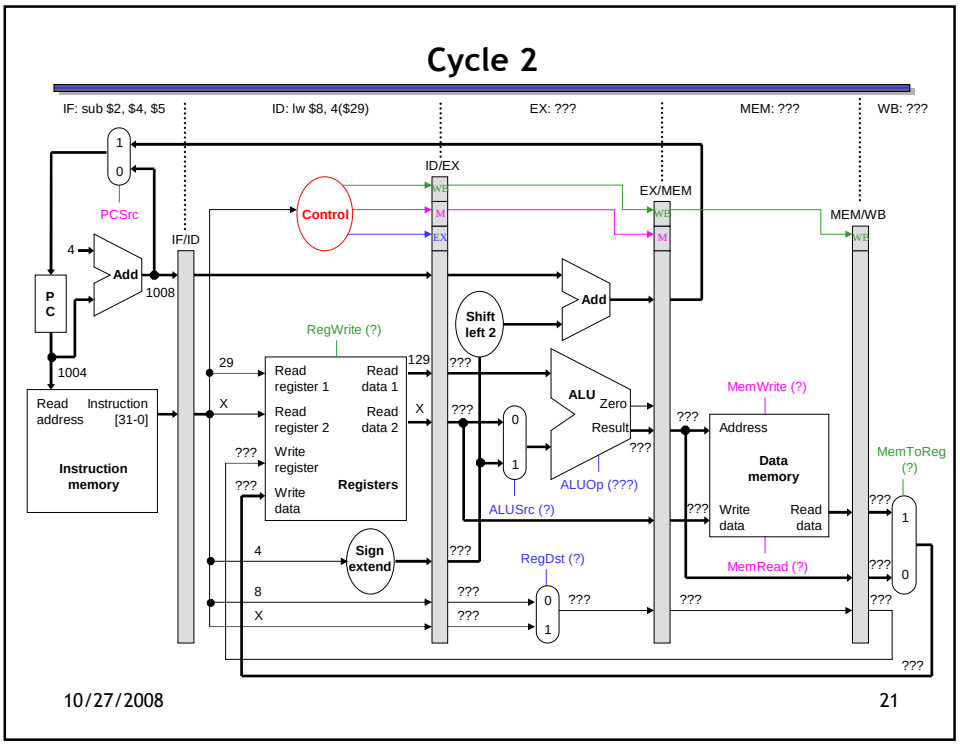
addresses in decimal

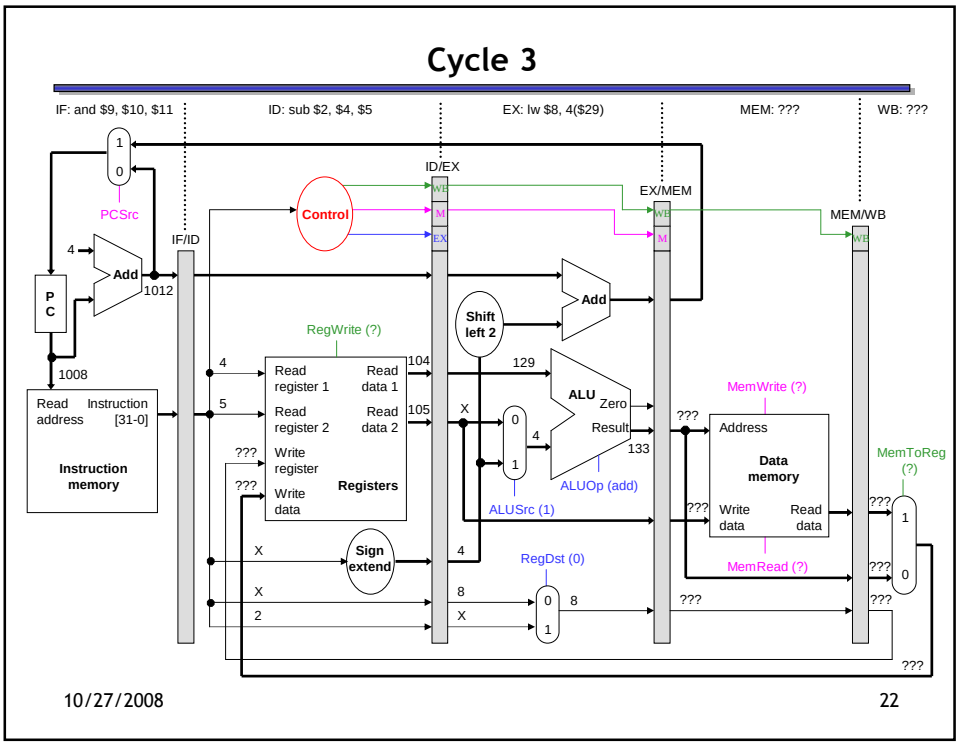
```
1000:  lw  $8, 4($29)
1004:  sub $2, $4, $5
1008:  and $9, $10, $11
1012:  or  $16, $17, $18
1016:  add $13, $14, $0
```

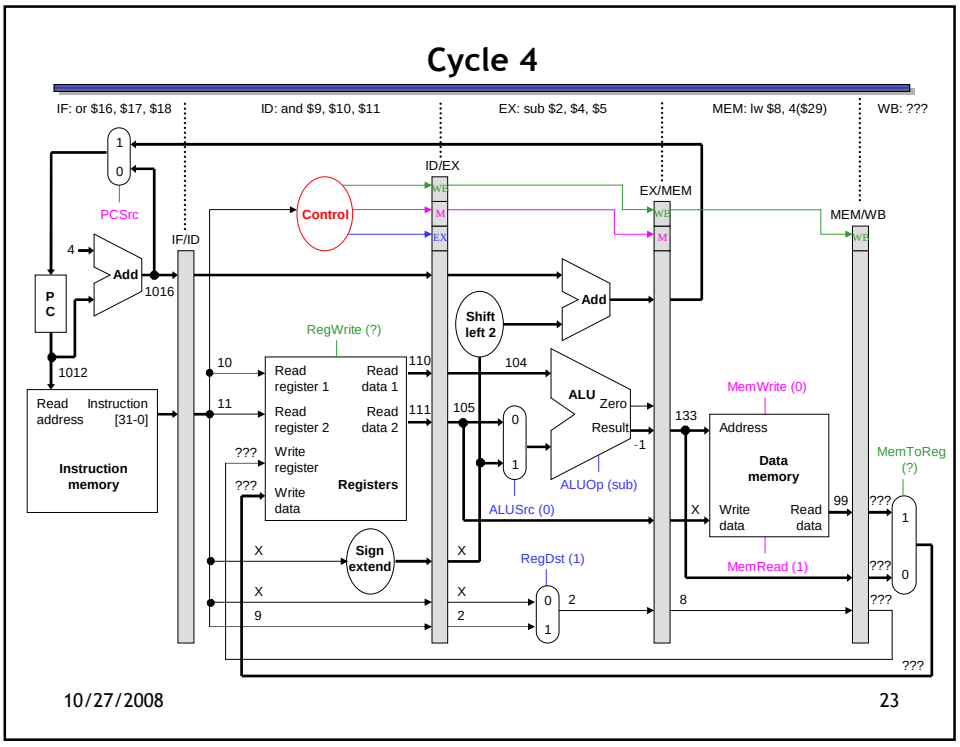
- We'll make some assumptions, just so we can show actual data values.
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - Every data memory location contains 99.
- Our pipeline diagrams will follow some conventions.
  - An X indicates values that aren't important, like the constant field of an R-type instruction.
  - Question marks ??? indicate values we don't know, usually resulting from instructions coming before and after the ones in our example.

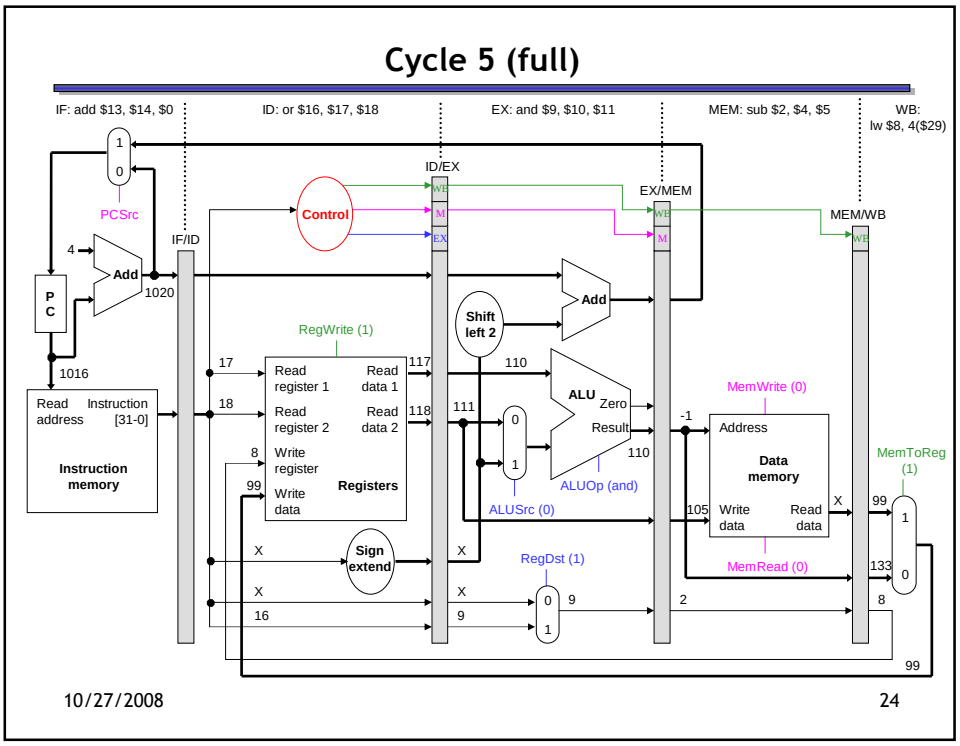
Cycle 1 (filling)



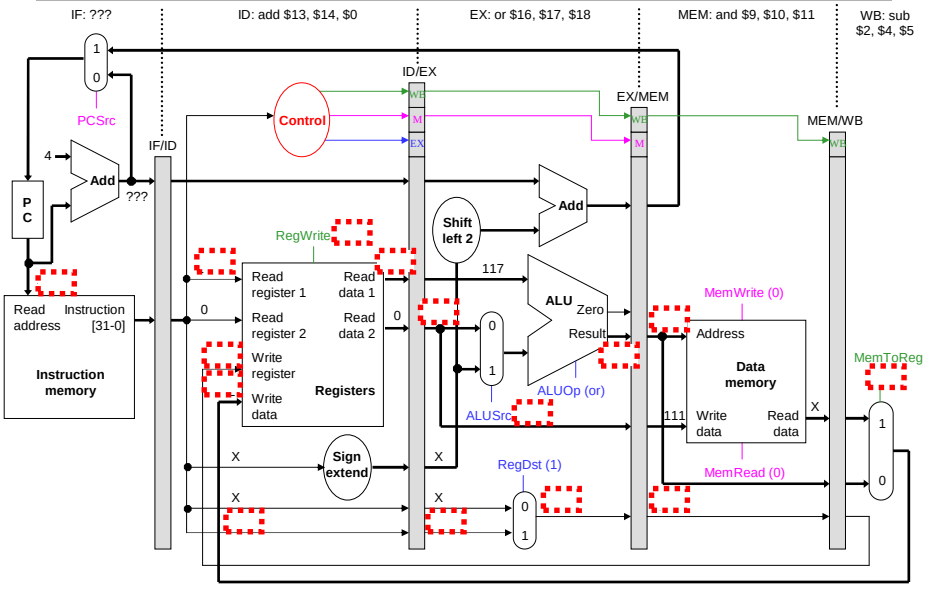

Cycle 2
IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
1008
1004
P C
Shift left 2
RegWrite (?)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Instruction memory
Read address
Instruction [31-0]
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (?)
RegDst (?)
Sign extend
10/27/2008
21


Cycle 3
IF: and $9, $10, $11
ID: sub $2, $4, $5
EX: lw $8, 4($29)
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
1012
1008
P C
Shift left 2
RegWrite (?)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Instruction memory
Read address
Instruction [31-0]
ALU
Zero
Result
133
ALUOp (add)
ALUSrc (1)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (?)
RegDst (0)
Sign extend
10/27/2008
22

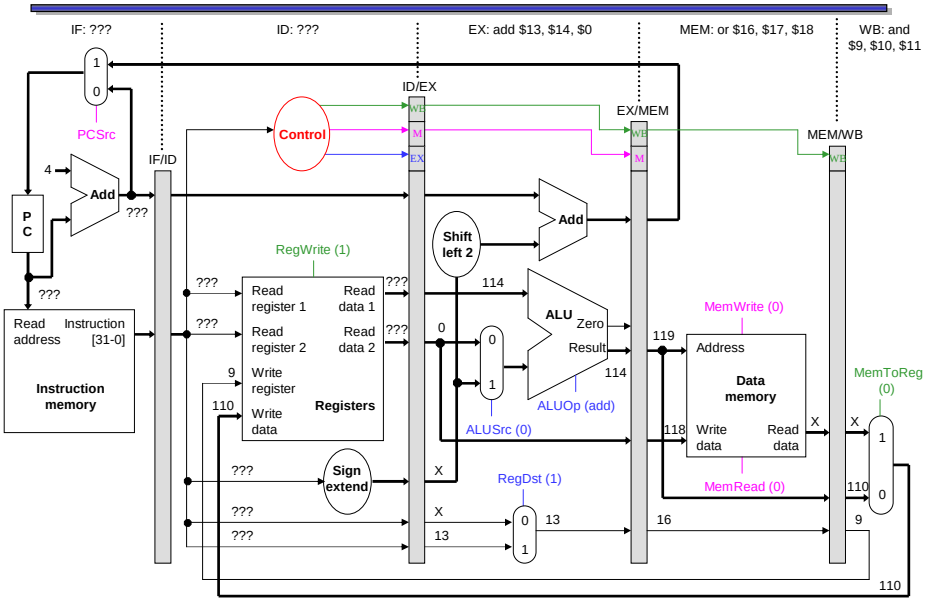

Cycle 4
IF: or $16, $17, $18
ID: and $9, $10, $11
EX: sub $2, $4, $5
MEM: lw $8, 4($29)
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
Shift left 2
RegWrite (?)
MemWrite (0)
MemToReg (?)
ALU
Zero
Result
ALUOp (sub)
ALUSrc (0)
RegDst (1)
MemRead (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
Address
Data memory
Read data
Write data
Instruction memory
Read address
Instruction [31-0]
P C
10/27/2008
23

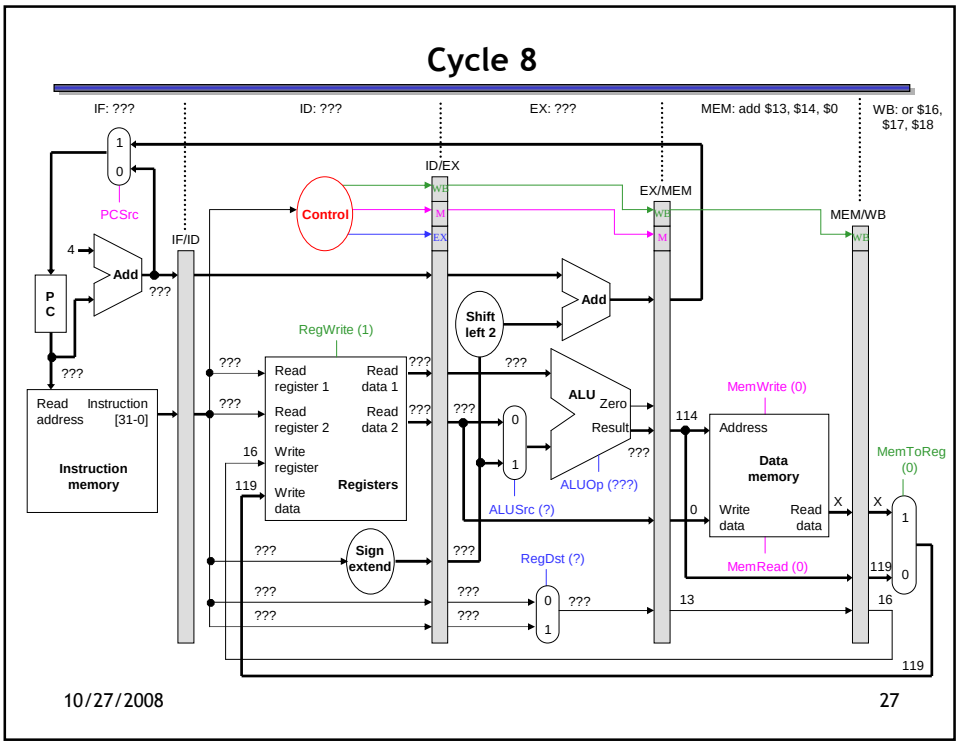

Cycle 5 (full)
IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
Shift left 2
RegWrite (1)
MemWrite (0)
MemToReg (1)
ALU
Zero
Result
ALUOp (and)
ALUSrc (0)
RegDst (1)
MemRead (0)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
Address
Data memory
Read data
Write data
Instruction memory
Read address
Instruction [31-0]
P C
10/27/2008
24

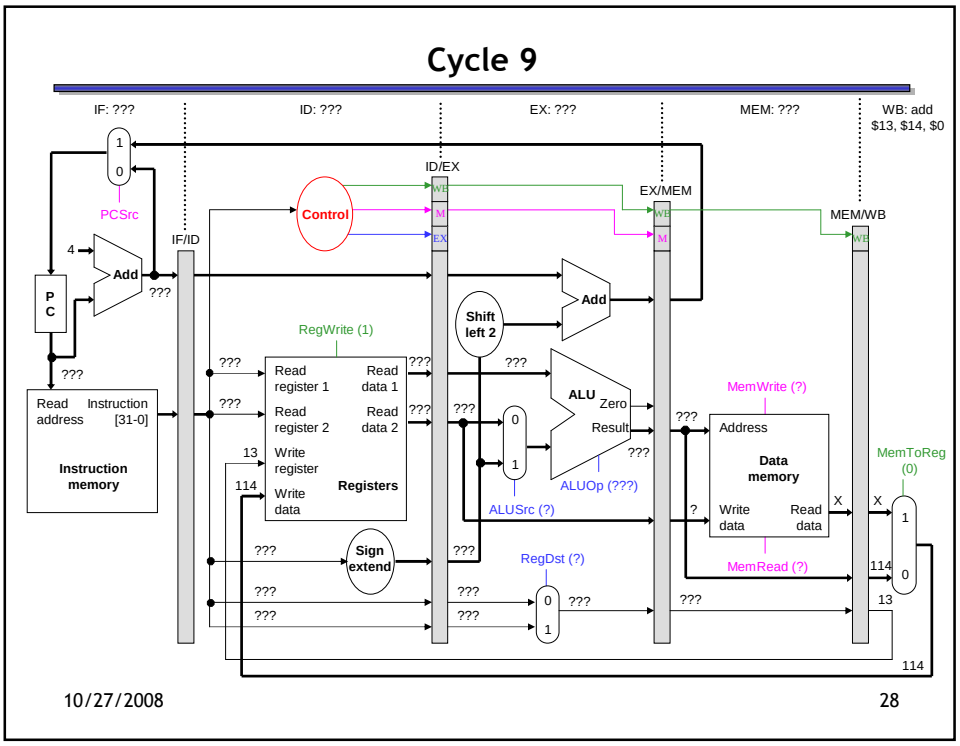
Cycle 6 (emptying)



Cycle 7




Cycle 8
IF: ???
ID: ???
EX: ???
MEM: add $13, $14, $0
WB: or $16, $17, $18
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
P C
Instruction memory
Read address
Instruction [31-0]
RegWrite (1)
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Shift left 2
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
Sign extend
RegDst (?)
MemWrite (0)
MemRead (0)
Data memory
Address
Write data
Read data
MemToReg (0)
10/27/2008
27


Cycle 9
IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
Add
P C
Instruction memory
Read address
Instruction [31-0]
RegWrite (1)
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Shift left 2
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
Sign extend
RegDst (?)
MemWrite (?)
MemRead (?)
Data memory
Address
Write data
Read data
MemToReg (0)
10/27/2008
28

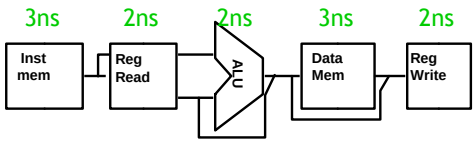
That's a lot of diagrams there

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$t5, $t6, $0					IF	ID	EX	MEM	WB

- Compare the last nine slides with the pipeline diagram above.
  - You can see how instruction executions are overlapped.
  - Each functional unit is used by a *different* instruction in each cycle.
  - The pipeline registers save control and data values generated in previous clock cycles for later use.
  - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

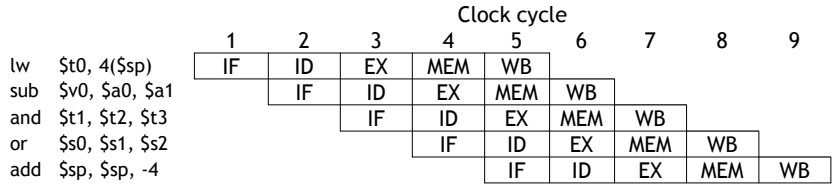
Performance Revisited

- Assuming the following functional unit latencies:



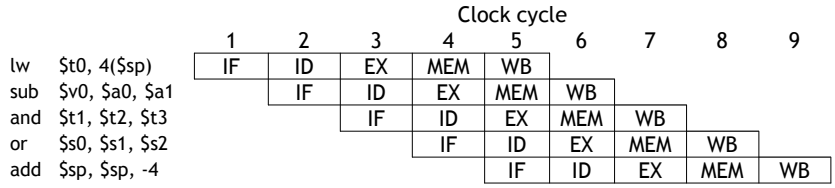
- What is the cycle time of a single-cycle implementation?
  - What is its throughput?
- What is the cycle time of a ideal pipelined implementation?
  - What is its steady-state throughput?
- How much faster is pipelining?

Ideal speedup



- In our pipeline, we can execute up to five instructions simultaneously.
  - This implies that the maximum speedup is 5 times.
  - In general, the ideal speedup equals the pipeline depth.
- Why was our speedup on the previous slide "only" 4 times?
  - The pipeline stages are imbalanced: a register file and ALU operations can be done in 2ns, but we must stretch that out to 3ns to keep the ID, EX, and WB stages synchronized with IF and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

The pipelining paradox



- Pipelining does *not* improve the execution time of any single instruction. Each instruction here actually takes *longer* to execute than in a single-cycle datapath (15ns vs. 12ns)!
- Instead, pipelining increases the throughput, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.
- The result is improved execution time for a *sequence* of instructions, such as an entire program.

Instruction set architectures and pipelining

- The MIPS instruction set was designed especially for easy pipelining.
  - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle.
  - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage.
  - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler.
- Pipelining is harder for older, more complex instruction sets.
  - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields.
  - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage.

Summary

- The pipelined datapath combines ideas from the single and multicycle processors that we saw earlier.
  - It uses multiple memories and ALUs.
  - Instruction execution is split into several stages.
- Pipeline registers propagate data and control values to later stages.
- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.
- Next lecture, we'll start talking about Hazards.