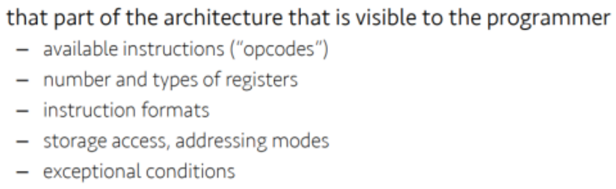
Computer Architecture

Tuesday, March 29, 2022 3:19 PM

Computer Architecture:

Instruction Set Architecture: How a user talks to the machine

- The agreed upon interface between all software that runs on the machine and hardware that executes it.



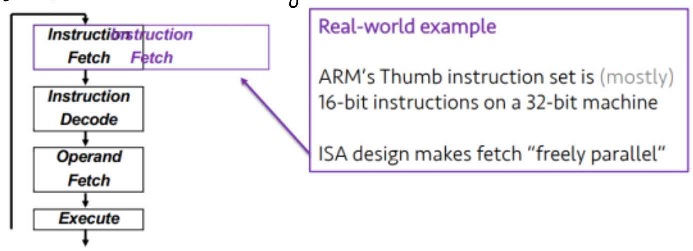
that part of the architecture that is visible to the programmer

- available instructions ("opcodes")
- number and types of registers
- instruction formats
- storage access, addressing modes
- exceptional conditions

Machine Organization: How the machine looks

Cycle: smallest unit of time in a processor

Parallelism: ability to do multiple things at once

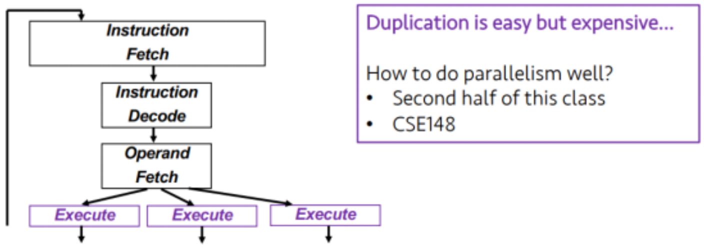


Real-world example

ARM's Thumb instruction set is (mostly) 16-bit instructions on a 32-bit machine

ISA design makes fetch "freely parallel"

Superscalar: execute more than one instruction per cycle

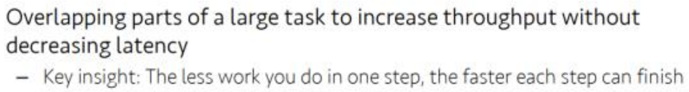


Duplication is easy but expensive...

How to do parallelism well?

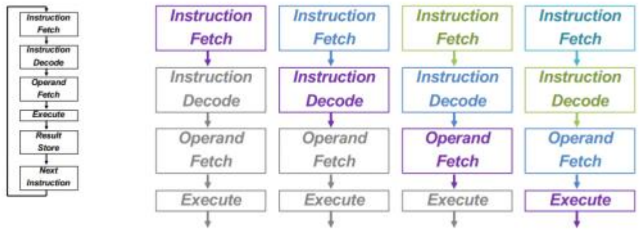
- Second half of this class
- CSE148

Pipelining: overlap tasks to increase throughput w/out increasing latency



Overlapping parts of a large task to increase throughput without decreasing latency

- Key insight: The less work you do in one step, the faster each step can finish



Designing ISA

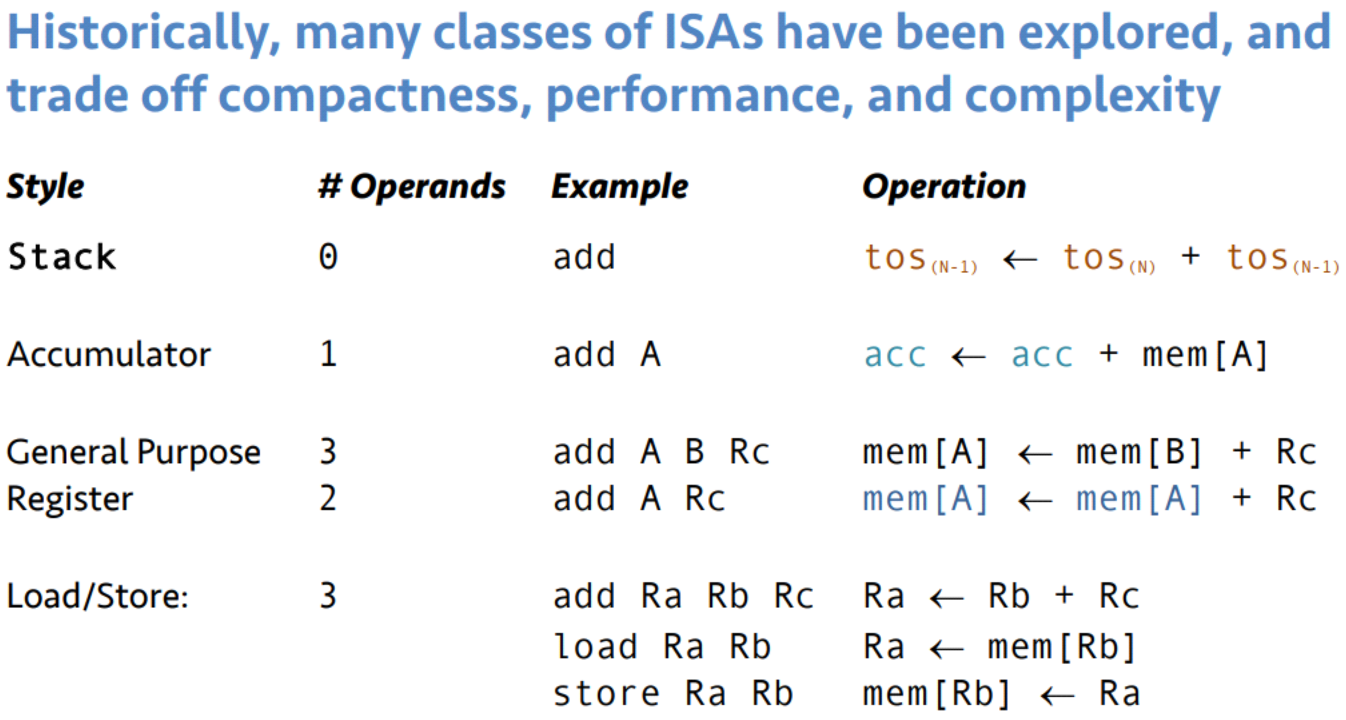
Thursday, March 31, 2022 3:35 PM

Key Questions:

- Operations
    - How many?
    - Which ones?
- Operands
    - How many?
    - Location
    - Types
    - How to Specify?
- Instruction Format
    - Size (bits)
    - How many formats?

History of ISA, Comparing ISA

Thursday, March 31, 2022 4:10 PM

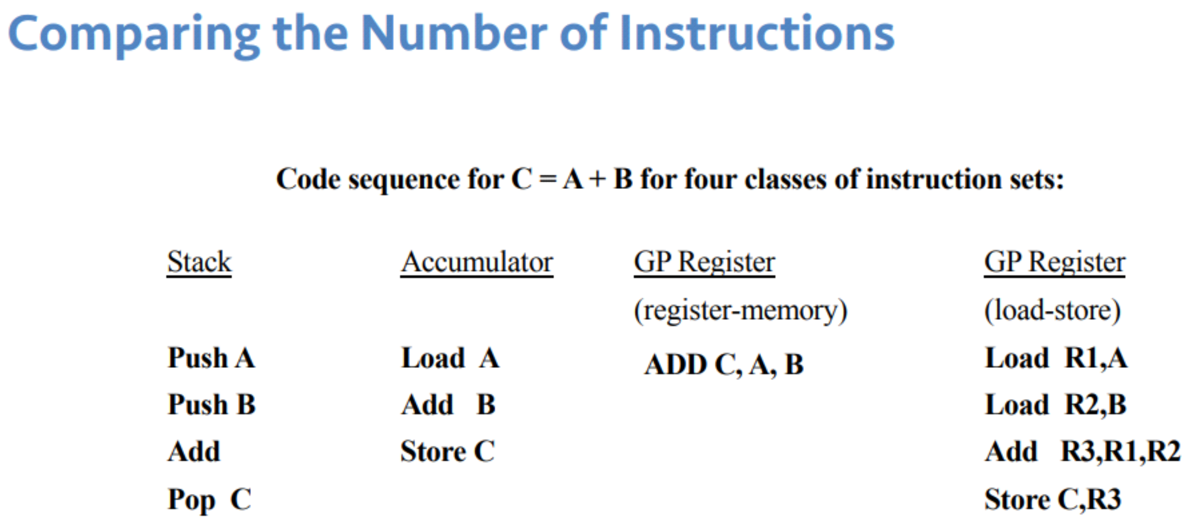


Historically, many classes of ISAs have been explored, and trade off compactness, performance, and complexity

Style	***# Operands***	***Example***	***Operation***
Stack	0	add	$tos_{(N-1)} \leftarrow tos_{(N)} + tos_{(N-1)}$
Accumulator	1	add A	acc ← acc + mem[A]

General Purpose Register	3
	2
		add A B Rc
		add A Rc
			mem[A] ← mem[B] + Rc
			mem[A] ← mem[A] + Rc

Load/Store:	3	add Ra Rb Rc
		load Ra Rb
		store Ra Rb
			Ra ← Rb + Rc
			Ra ← mem[Rb]
			mem[Rb] ← Ra



Comparing the Number of Instructions

Code sequence for C = A + B for four classes of instruction sets:

Stack	Accumulator	GP Register (register-memory)	GP Register (load-store)
Push A	**Load A**	**ADD C, A, B**	**Load R1,A**
Push B	**Add B**		**Load R2,B**
Add	**Store C**		**Add R3,R1,R2**
Pop C			**Store C,R3**

MIPS Design, Instruction Formats, Addressing Modes

Thursday, March 31, 2022 3:46 PM

- Fixed Length Instructions (MIPS)
  - Easy fetch and decode
  - Simplify pipelining and parallelism
- Variable-length instructions (x86, VAX)
  - Multi-step fetch and decode
  - Much more flexible and compact instruction set
- Hybrid instructions (ARM)
  - Middle ground

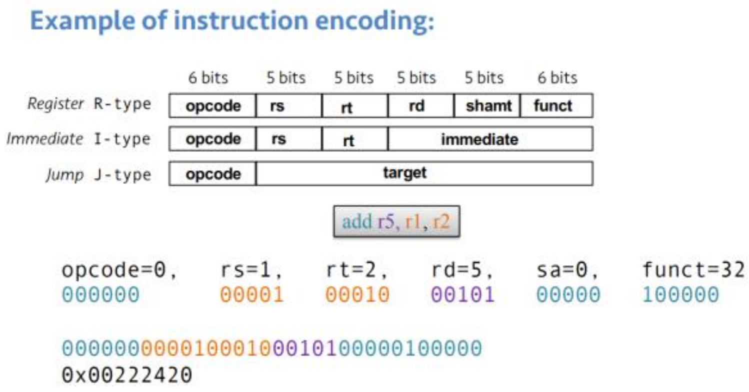
MIPS Instructions are 32 bits long

- Many different instruction formats
  - Complicates decoding
  - Uses more instruction bits to specify format
  - Allow usage of variable length ISA

MIPS has 3 instruction formats, fixed 32 bit instruction size

- Operands
  - Registers (32 options)
  - Memory (2^32 locations)
- Registers are easy to specify, close to processor (fast access)
- Load-store architectures
  - Normal arithmetic instructions only use registers
  - Access memory only with explicit load/store instructions

MIPS most arithmetic instructions have 3 operands



Example of instruction encoding:

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Register R-type	opcode	rs	rt	rd	shamt	funct
Immediate I-type	opcode	rs	rt	immediate		
Jump J-type	opcode	target				

add r5, r1, r2

```
opcode=0,  rs=1,  rt=2,  rd=5,  sa=0,  funct=32
000000     00001  00010  00101  00000  100000

00000000001000100010100000100000
0x00222420
```

- Addressing modes
  - Register direct - R3
  - Immediate (literal) - #25
  - Direct (absolute) - M[1000]
  - Register indirect - M[R3]
  - Base Displacement - M[R3 + 1000]
  - Base Index - M[R3 + R4]
  - Scaled Index - M[R3 + R4 * d + 1000]
  - Autoincrement - M[R3++]
  - Autodecrement - M[R3--]
  - Memory Indirect - M[M[R3]]

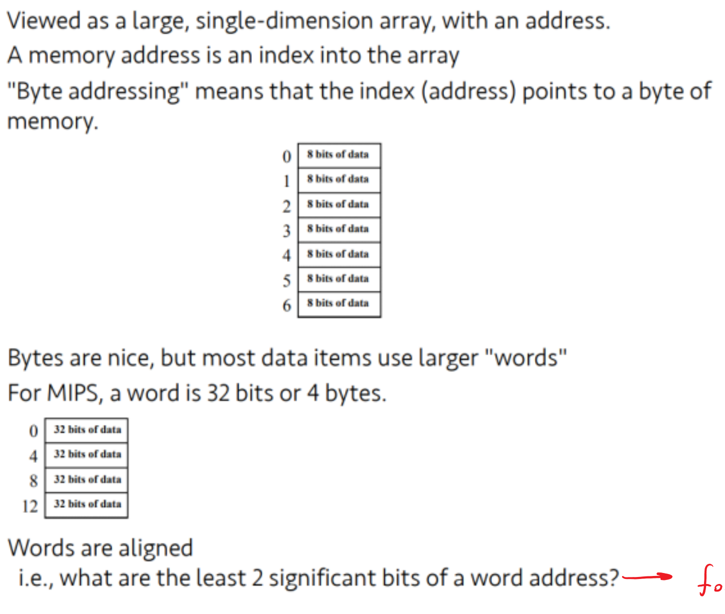
MIPS uses Register direct, Immediate, Base Displacement

Memory Structure

Thursday, March 31, 2022 5:24 PM

Memory can be represented as an array of bytes

MIPS is a 32 bit word architecture, each instruction and data value is 32 bits, or 4 bytes



Viewed as a large, single-dimension array, with an address.

A memory address is an index into the array

"Byte addressing" means that the index (address) points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

Bytes are nice, but most data items use larger "words"

For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

Words are aligned

i.e., what are the least 2 significant bits of a word address? → for MIPS, always 00 because words are divisable by 4

Control Flow, MIPS Jump/Branch Instructions

Tuesday, April 5, 2022 3:42 PM

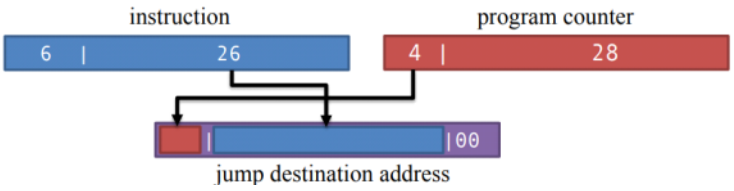
Jumps
- ○ Used to implement GOTO, initialization
- Procedure call (jump routine)
  - ○ Used to implement functions
- Conditional Branch
  - ○ Used to implement if-the-else, loops

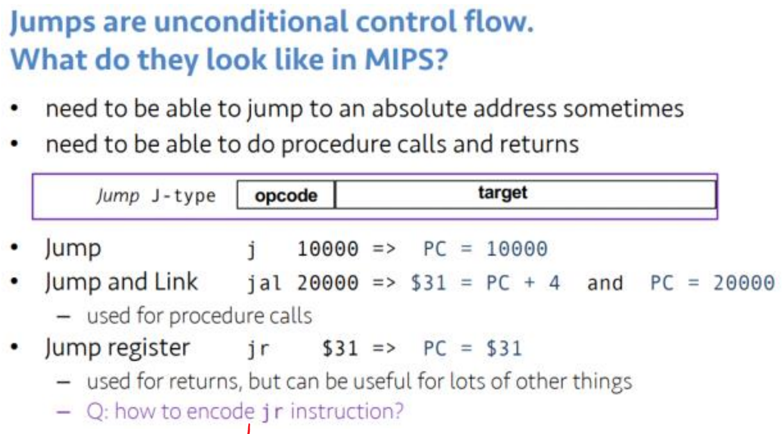
- Control flow specifies two things
  - ○ Condition to jump
  - ○ Location to jump to

Jump: j <Location>

Jump and link: jal <Location>
$31 = PC + 4

Jump register: jr $31
PC = $31





Jumps are unconditional control flow. What do they look like in MIPS?

- need to be able to jump to an absolute address sometimes
- need to be able to do procedure calls and returns

Jump J-type	opcode	target

- Jump `j 10000 => PC = 10000`
- Jump and Link `jal 20000 => $31 = PC + 4 and PC = 20000`
  - used for procedure calls
- Jump register `jr $31 => PC = $31`
  - used for returns, but can be useful for lots of other things
  - Q: how to encode jr instruction?

jr uses the R type format

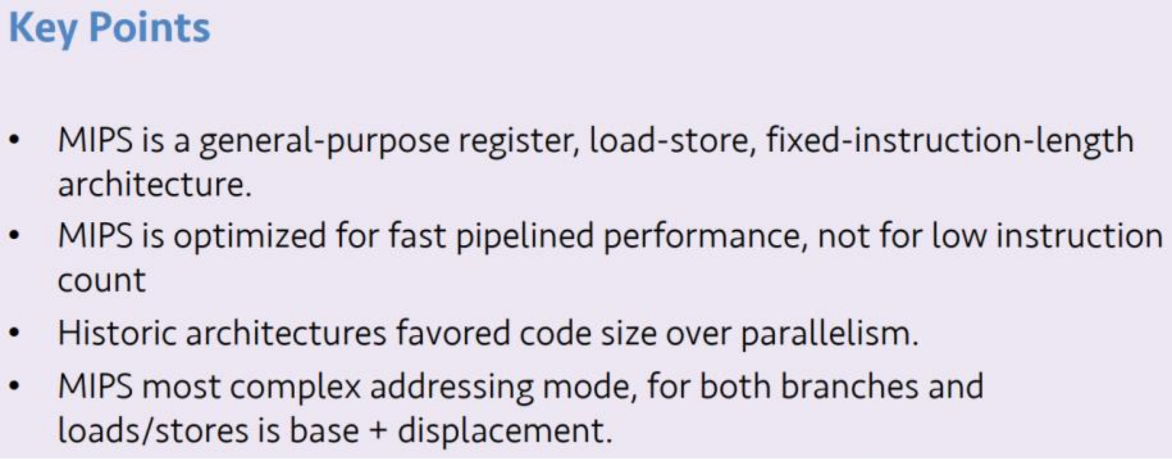
Branch on Equal: beq r1, r2, offset
PC = (PC + 4) + offset * 4

Branch on Not Equal: bne r1, r2, offset
PC = (PC + 4) + offset * 4

Store less than: slt $1, $2, $3 -> if ($2 < $3)
set $1 = 1, otherwise $1 = 0

MIPS Instructions

Tuesday, April 5, 2022 3:36 PM



Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Historic architectures favored code size over parallelism.
- MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.

Arithmetic:

- add, subtract, multiply, divide
- but NOT: mod, exponents, add with carry, sin, cos

Logical

- and, or, shift left, shift right, xor
- but NOT: nand, nor, bit clear

Data Transfer

- load word, store word, load half, store half
- but NOT: post increment load/store, direct operations on memory contents, load/store multiple

MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2 ^16	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equa	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Measuring Performance

Thursday, April 7, 2022 3:28 PM

- Time to do a task
  - Execution time, response time, latency
- Tasks per unit time
  - Throughput, bandwidth

- Ways to represent performance
  - Execution time
  - Throughput (operations / time)
  - Frame rate
  - Responsiveness
  - Performance / Cost
  - Performance / Power
  - Performance / Energy

- Ways to measure execution time
  - Program reported time?
  - Wall-clock time?
  - User CPU time?
  - User + Kernel CPU time?

$$\text{Performance}_X = \frac{1}{\text{Execution Time}_X} \text{ , for program X}$$

- Only has meaning in context of a specific program
- Not useful as absolute measurement, measures relative performance

$$\text{Speedup (X/Y)} = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = n$$

Where X is the experimental and Y is the baseline

Eg: A runs program C in 9s, B runs program C in 6s
Speedup(B / A) = 9 / 6 = 1.5 times faster

What is Time?

- CPU Execution Time = CPU clock cycles * Clock cycle time
- CPU clock cycles = number of instructions * (average) cycles per instruction

Execution Time = Instruction count * CPI * Clock cycle time

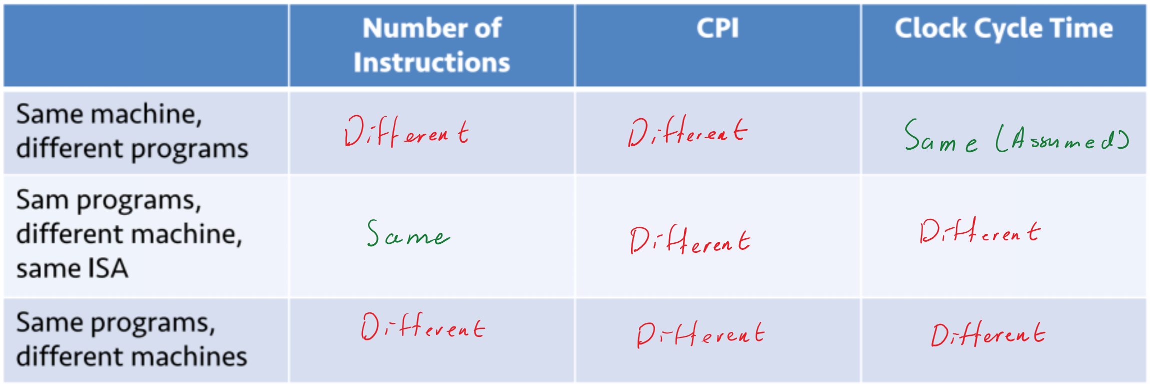
Modern machines can change cycle time/clock rate for efficiency

Who/What Affects Performance

Thursday, April 7, 2022 4:18 PM

Type	Instruction Count	CPI	Clock Cycle Time
Programmer	Yes	No	No
Compiler	Yes	Maybe (Optimization)	No
Instruction Set Architect	Yes	Yes	Yes
Machine Architect	No	Yes (RCA vs Carry Lookahead Adder)	Yes
Hardware Designer	No	No	Yes (change the critical delay through routing)
Material Physics	No	No	Maybe (change the property of materials)

CPU Execution Time = Instruction Count X CPI X Clock Cycle Time

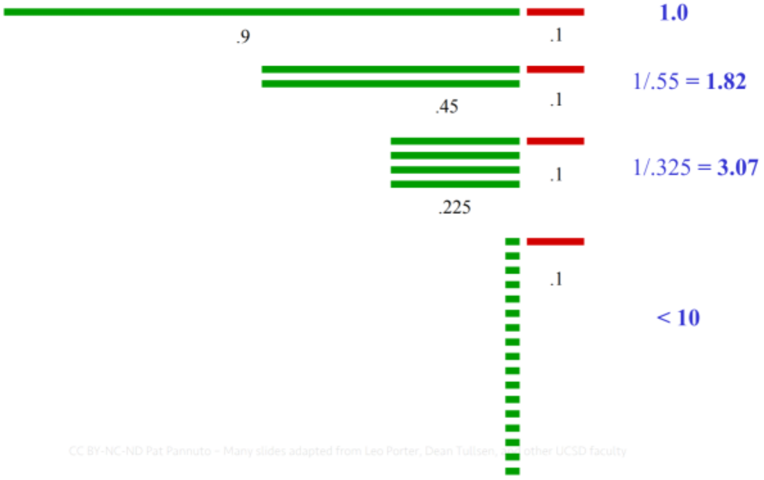


	Number of Instructions	CPI	Clock Cycle Time
Same machine, different programs	Different	Different	Same (Assumed)
Sam programs, different machine, same ISA	Same	Different	Different
Same programs, different machines	Different	Different	Different

Amdahl's Law, Parallelization

Thursday, April 7, 2022 4:32 PM

$$\text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}$$



N more cores does not mean it will be N times faster!

The red, unparallelizable portion of the workload limits the maximum performance improvement in parallelism.

Single Cycle Machines

Tuesday, April 12, 2022 3:19 PM

Idea: Each instruction takes exactly one cycle

- Advantage: One clock cycle per instruction
- Disadvantage: Long clock cycle

Idea: A single cycle machine's cycle time must be the time it takes for the slowest instruction.

Def: We will use a simplified MIPS instruction set

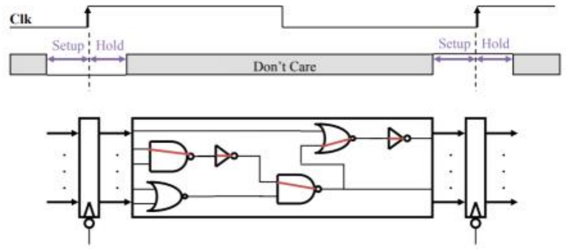
memory-reference instructions: `lw, sw`

arithmetic-logical instructions: `add, sub, and, or, slt`

control flow instructions: `beq`

Note: There is no multiply instructions because it is very slow

Review: Clock cycle time is dependent on the longest delay in a combinational path between storage elements

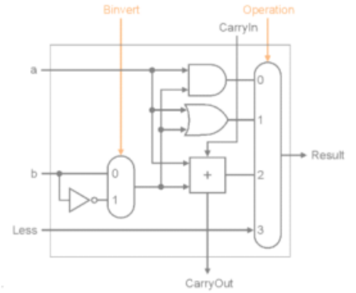


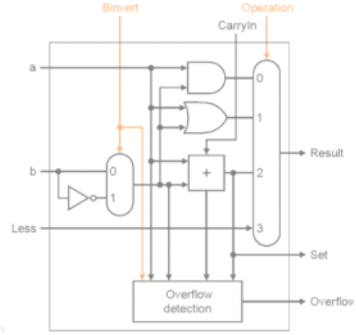
All storage elements are clocked by the same clock edge

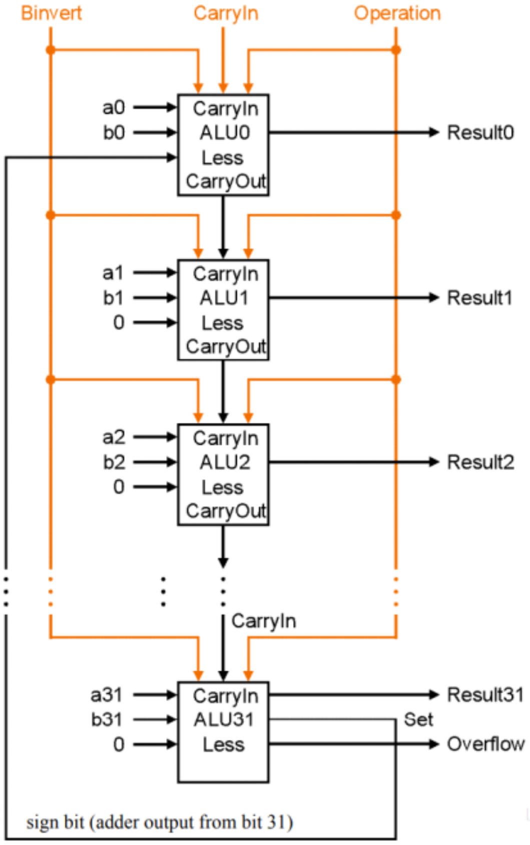
ALU Design

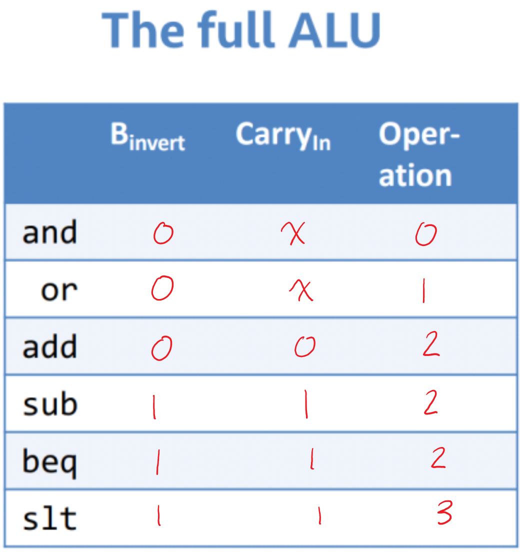
Tuesday, April 12, 2022 3:39 PM

Idea: Chain multiple 1-bit ALUs to create N-bit ALUs







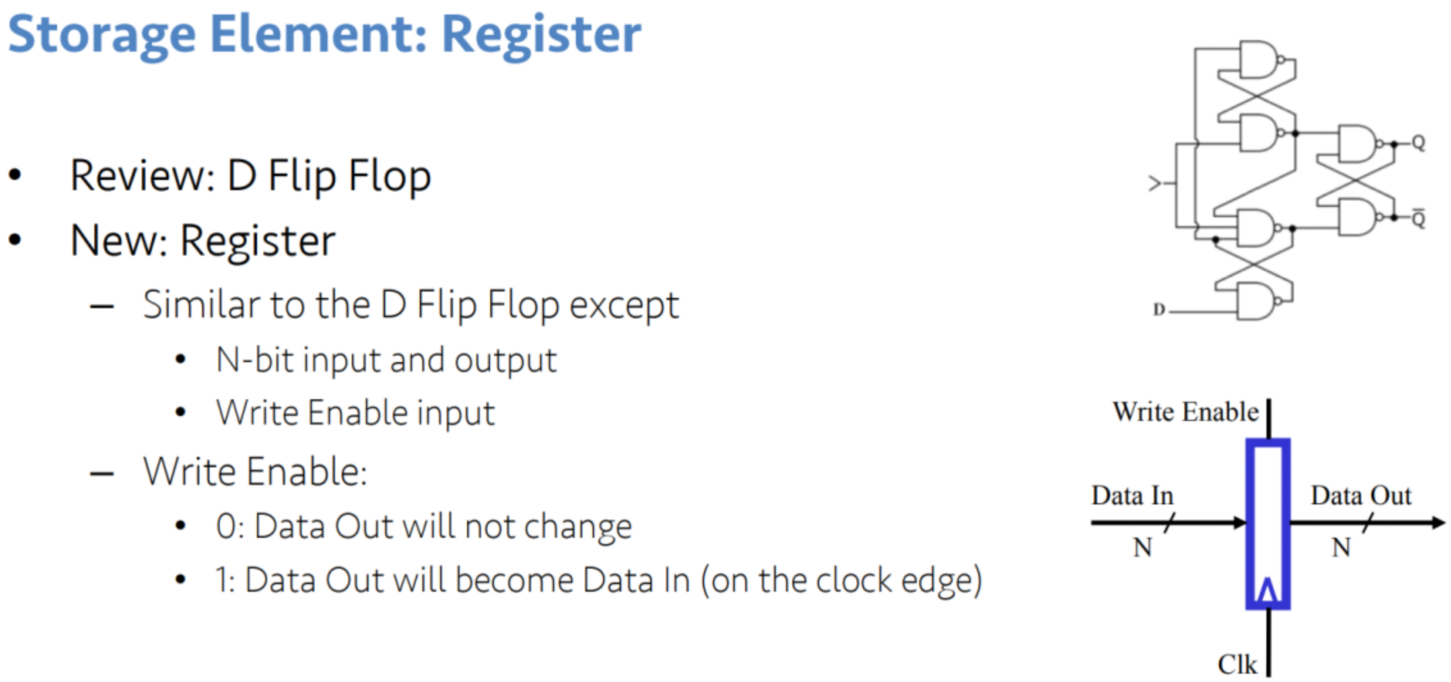


The full ALU

	B_{invert}	$Carry_{In}$	Oper-ation
and	0	X	0
or	0	X	1
add	0	0	2
sub	1	1	2
beq	1	1	2
slt	1	1	3

Registers, Register File

Tuesday, April 12, 2022 4:14 PM



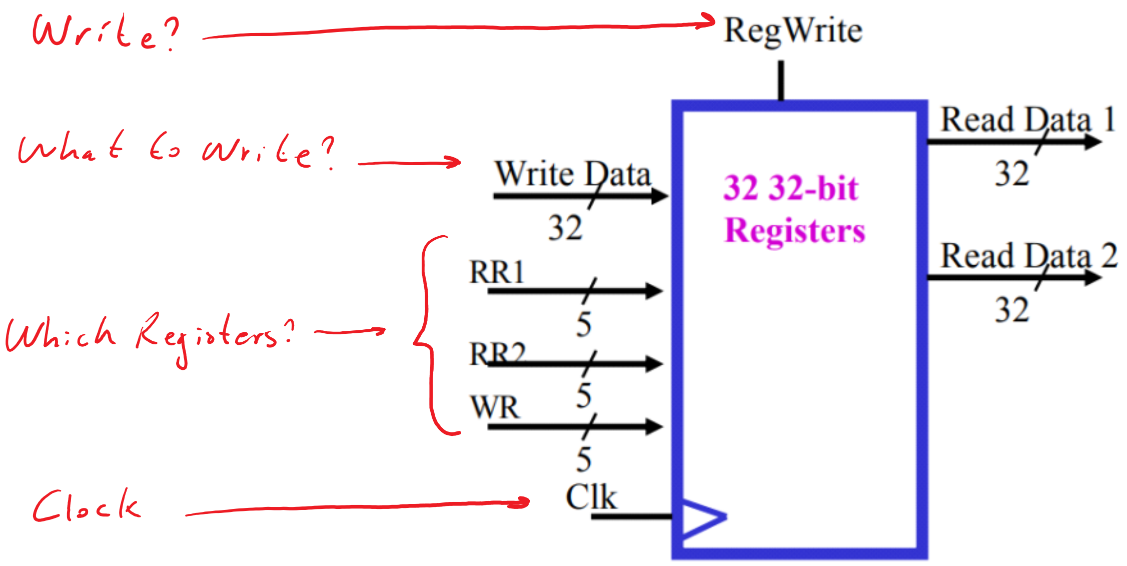
Storage Element: Register

- Review: D Flip Flop
- New: Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - 0: Data Out will not change
    - 1: Data Out will become Data In (on the clock edge)

Q
Q̄
D
Write Enable
Data In
Data Out
N
N
Clk

Idea: Combine many Registers into a Register File

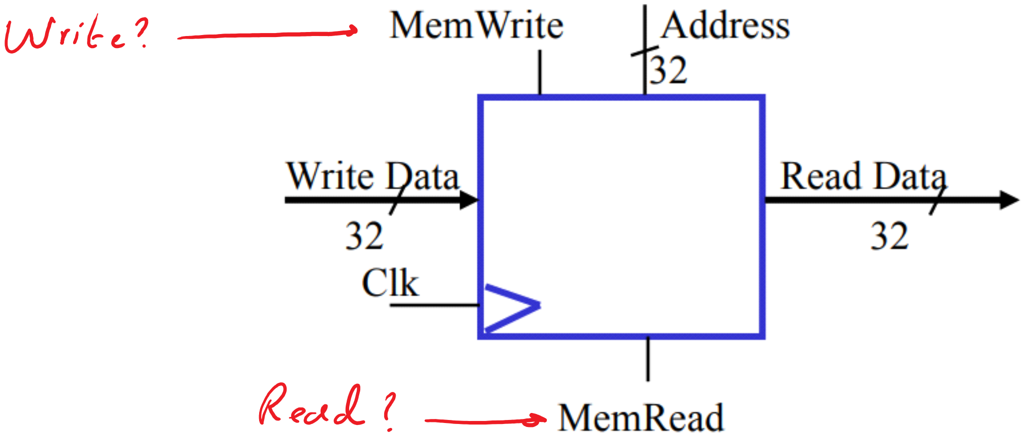
Def: A register file could look like:



Memory Interface

Tuesday, April 12, 2022 4:17 PM

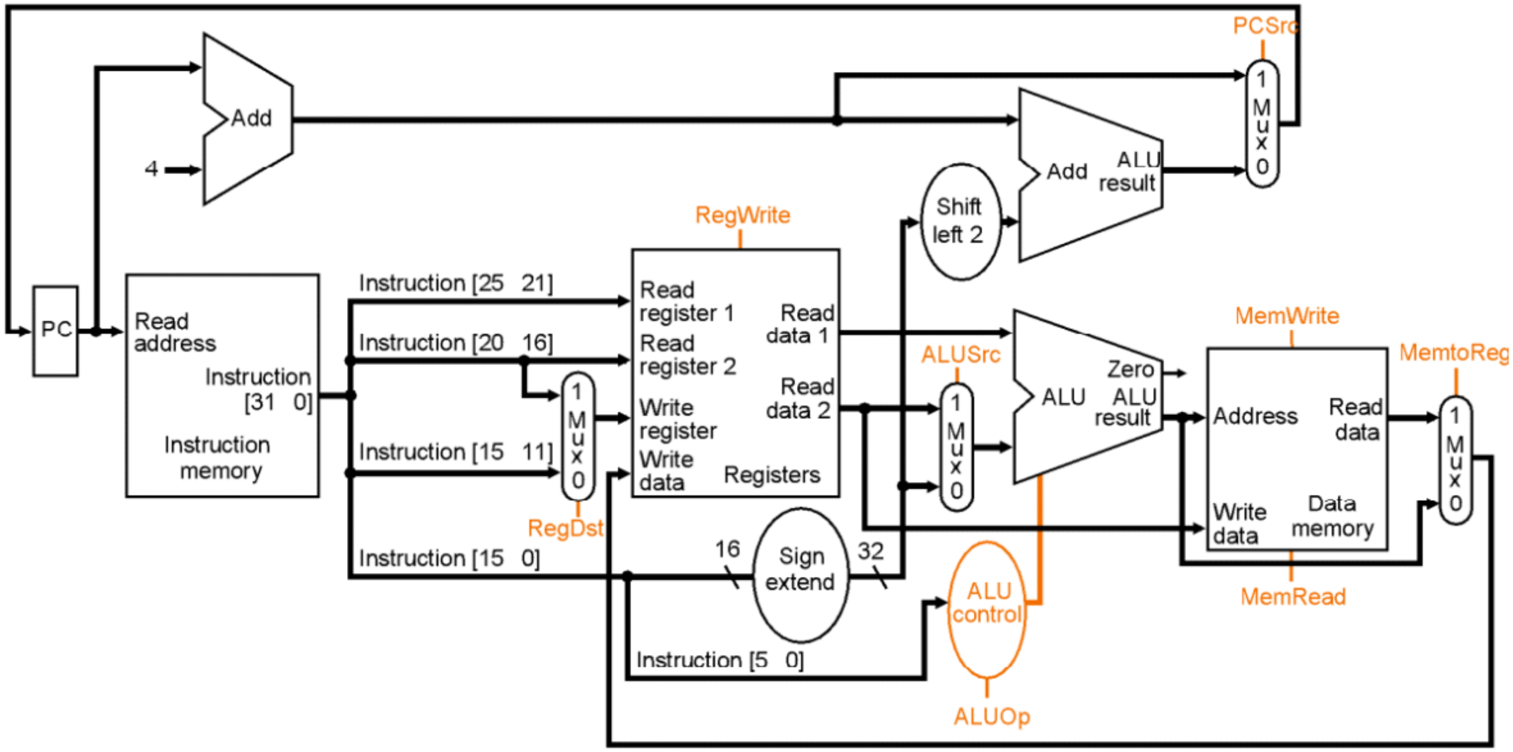
Def: A memory module might look like:



Datapath Design

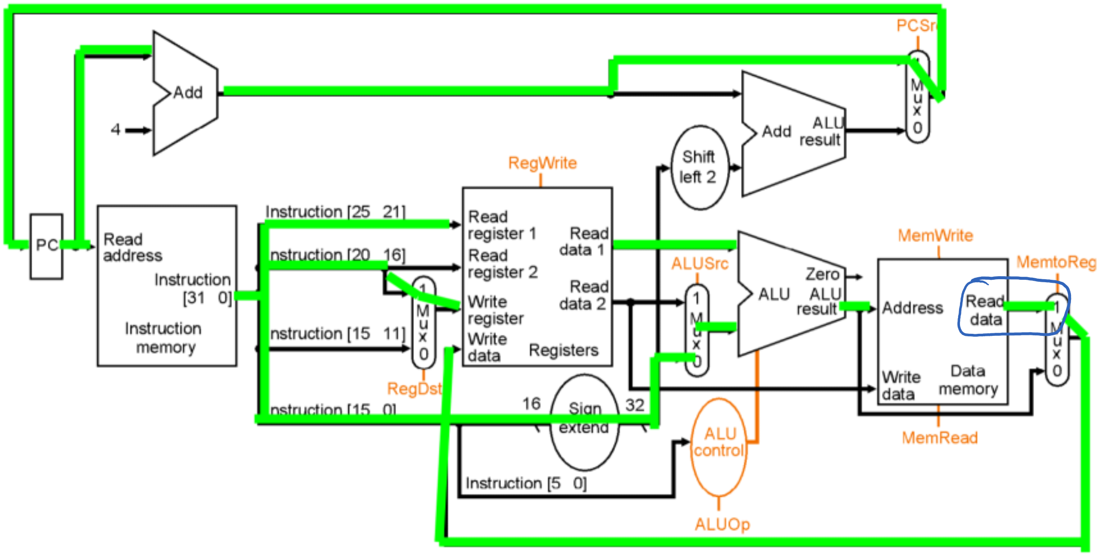
Thursday, April 14, 2022 3:28 PM

Def: A final design for the <u>datapath</u> might look like:



Understanding The Datapath Signals Examples

Thursday, April 14, 2022 3:39 PM



Ignoring control – which instruction does this active datapath represent

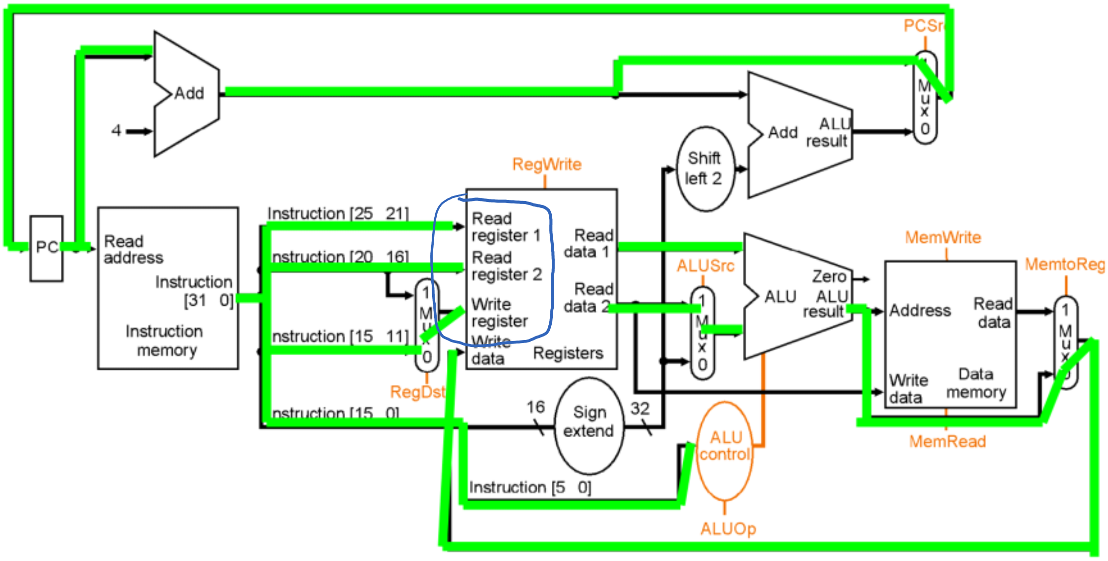
A. R-type

[✓ B.] lw

C. sw

D. Beq

E. None of the above



Ignoring control – which instruction does this active datapath represent

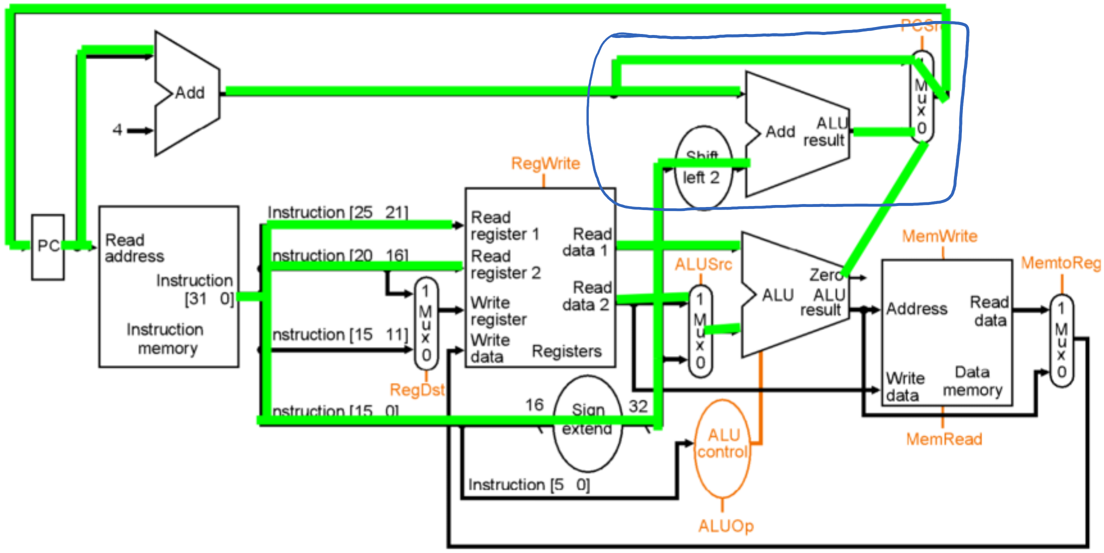
[✓ A.] R-type

B. lw

C. sw

D. Beq

E. None of the above



Ignoring control – which instruction does this active datapath represent

A. R-type

B. lw

C. sw

[✓ D.] Beq

E. None of the above

Ignoring control – which instruction does this active datapath represent

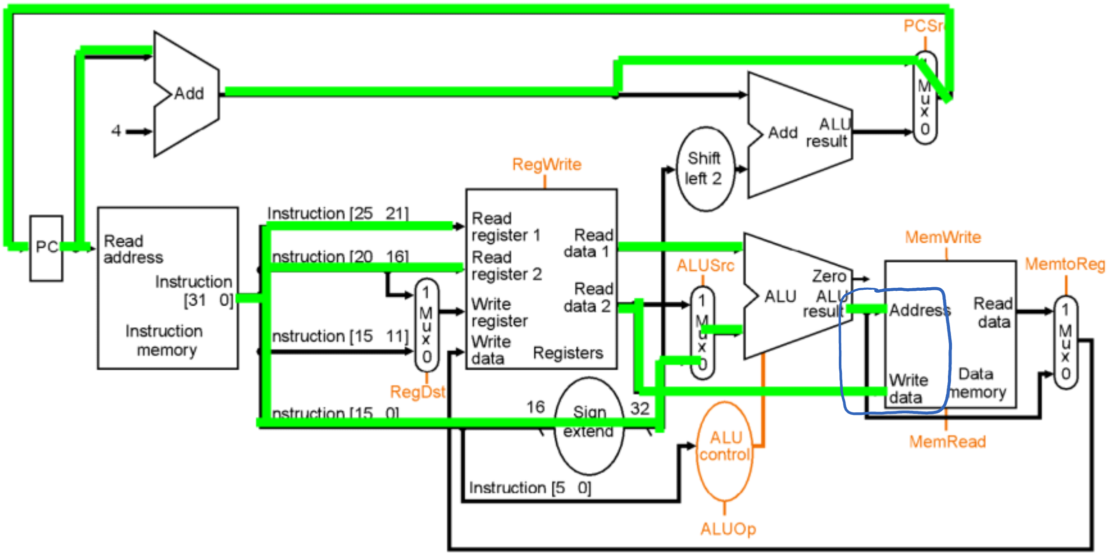
A. R-type

B. lw

[✓ C.] sw

D. Beq

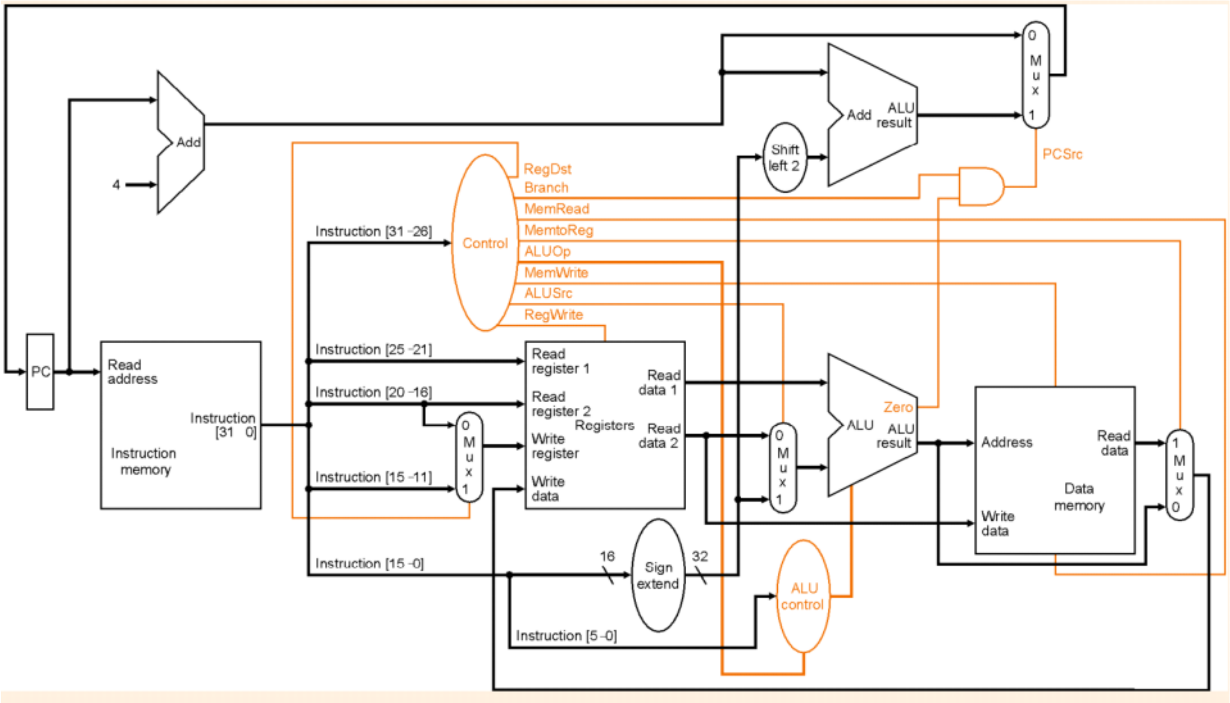
E. None of the above



Control Path Signals

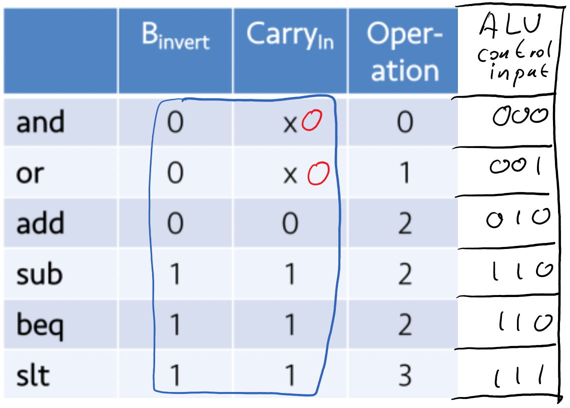
Thursday, April 14, 2022 3:47 PM

The signals in red are used to decide how the Datapath operates, information taken from opcode:

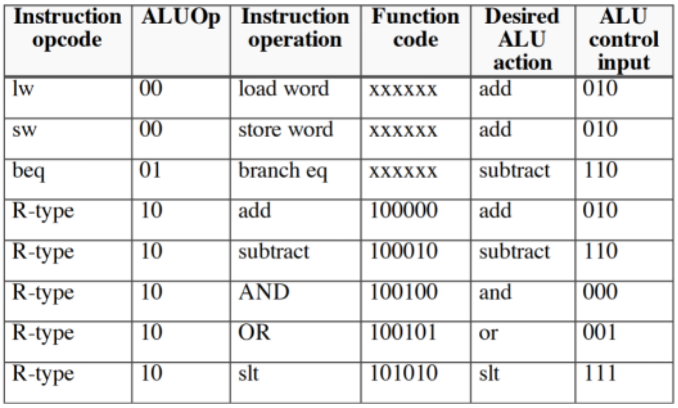


ALU Control:

Recall:

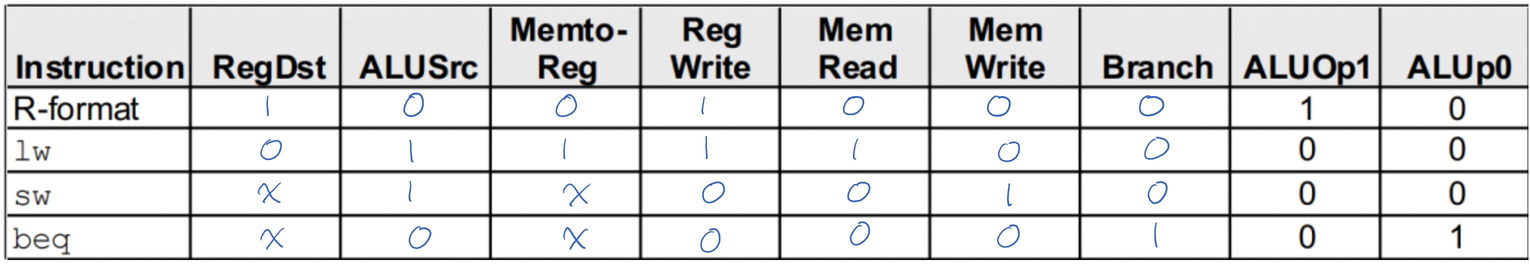


	B_{invert}	$Carry_{In}$	Oper-ation	ALU Control input
and	0	x 0	0	000
or	0	x 0	1	001
add	0	0	2	010
sub	1	1	2	110
beq	1	1	2	110
slt	1	1	3	111



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
beq	01	branch eq	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	slt	101010	slt	111

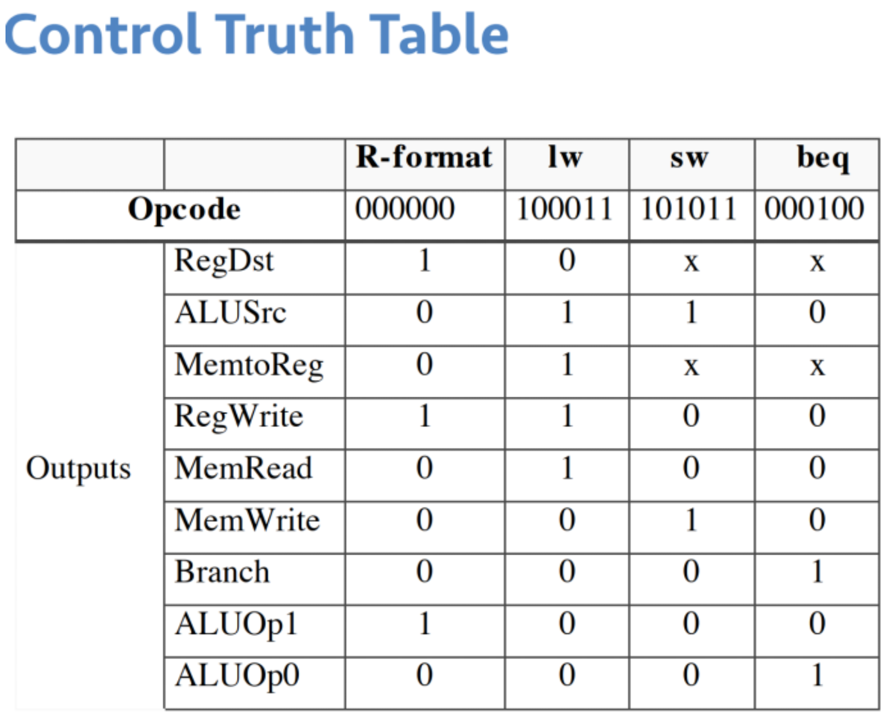
Control Unit:



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

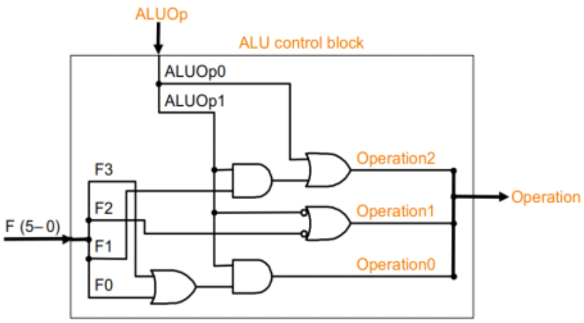
Control Path Summary

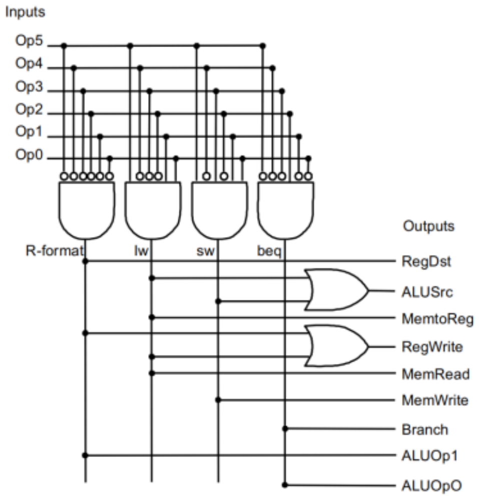
Thursday, April 14, 2022 4:37 PM



Control Truth Table

		R-format	lw	sw	beq
Opcode		000000	100011	101011	000100
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1





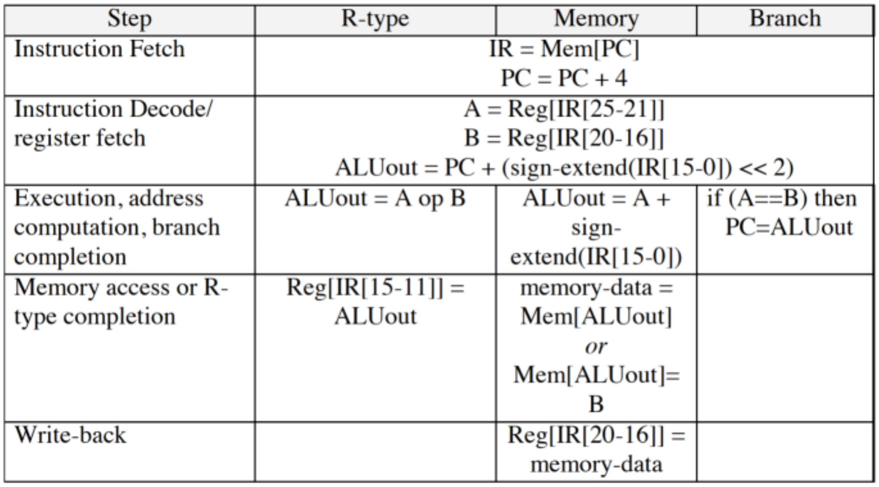
Multiple Cycle Machines

Thursday, April 14, 2022 4:41 PM

Problem: Some instructions may take much longer than other instructions
Idea: Break large instructions into smaller tasks, each one taking one cycle

It's essentially the same Datapath as the single cycle, but we use registers to store intermediate step

Execution steps:



Step	R-type	Memory	Branch

Instruction Fetch	IR = Mem[PC]
	PC = PC + 4		

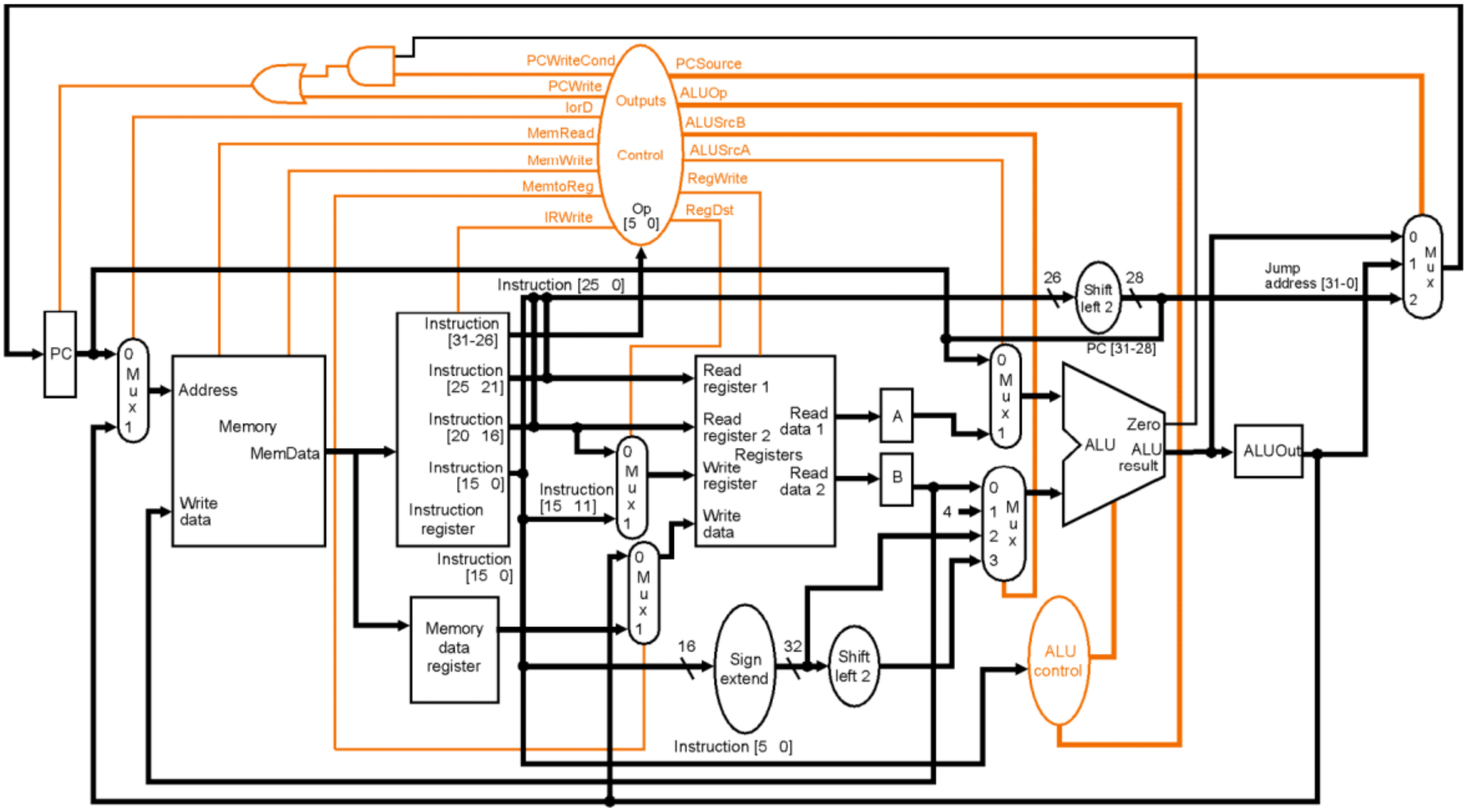
Instruction Decode/ register fetch	A = Reg[IR[25-21]]
	B = Reg[IR[20-16]]
	ALUout = PC + (sign-extend(IR[15-0]) << 2)		

Execution, address computation, branch completion	ALUout = A op B	ALUout = A + sign-extend(IR[15-0])	if (A==B) then PC=ALUout

Memory access or R-type completion	Reg[IR[15-11]] = ALUout	memory-data = Mem[ALUout]
		or
		Mem[ALUout]= B	

Write-back		Reg[IR[20-16]] = memory-data	

A multi cycle machine might look like:

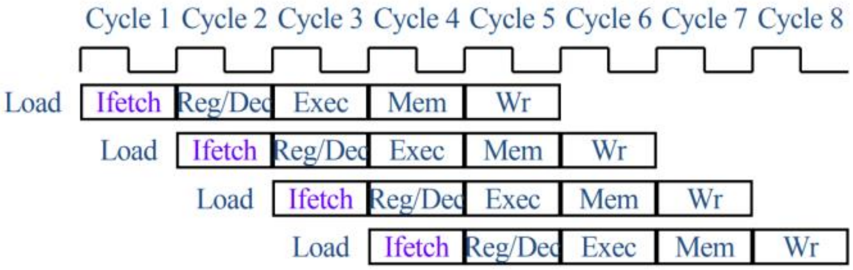


Pipelines, Pipelined Machines

Tuesday, April 19, 2022 3:29 PM

Idea: we can break an instruction into multiple tasks, and we can pipe tasks to increase throughput

Example: if an instruction has tasks: instruction fetch -> decode, register fetch -> execute -> memory access -> write back
Then a pipelined machine has the latency and throughput:



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
Load	Ifetch	Reg/Dec	Exec	Mem	Wr			
	Load	Ifetch	Reg/Dec	Exec	Mem	Wr		
		Load	Ifetch	Reg/Dec	Exec	Mem	Wr	
			Load	Ifetch	Reg/Dec	Exec	Mem	Wr

Higher maximum throughput, but control logic is more complicated

The Execution time = instructions * 1 * CT, the CPI is 1 because an instruction is completed every cycle

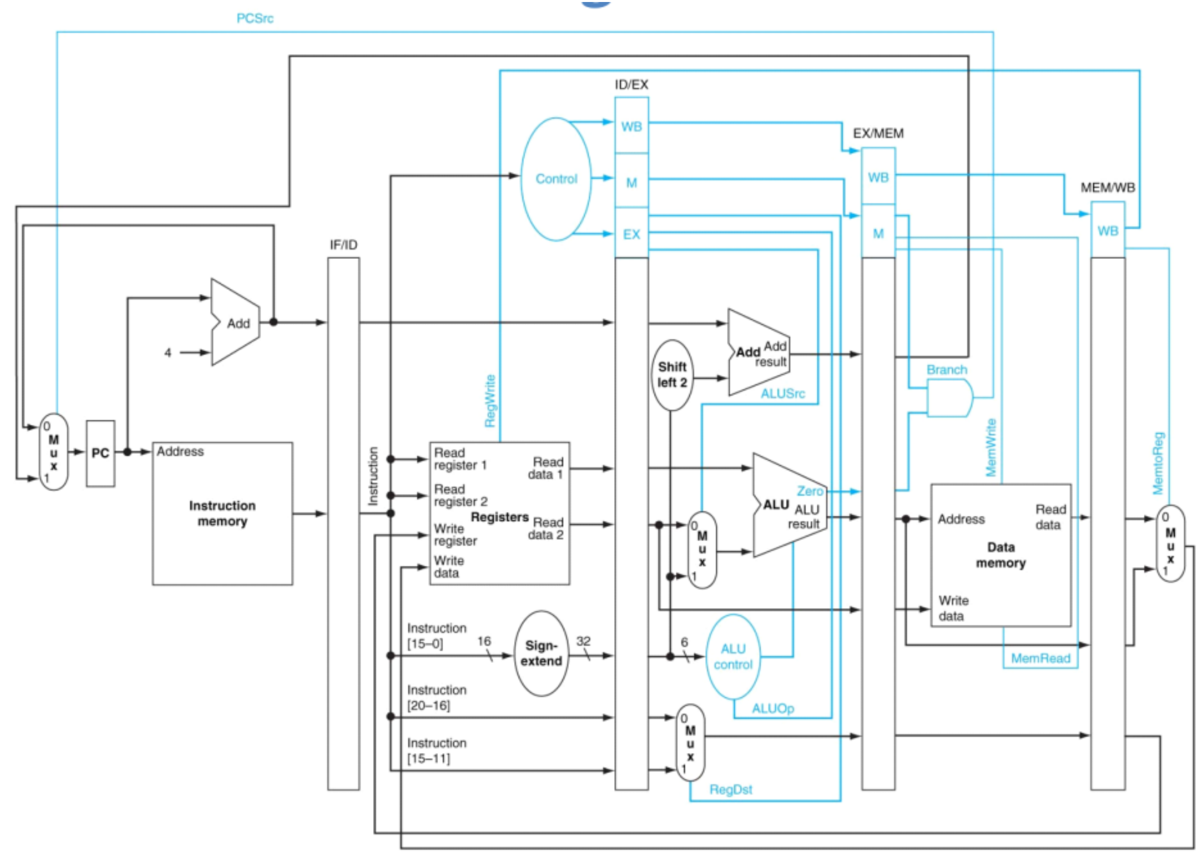
Note: to avoid conditions where a piece of hardware is used by multiple instructions at the same time, all instructions should have the same stages in the same order

Control Signals, Design

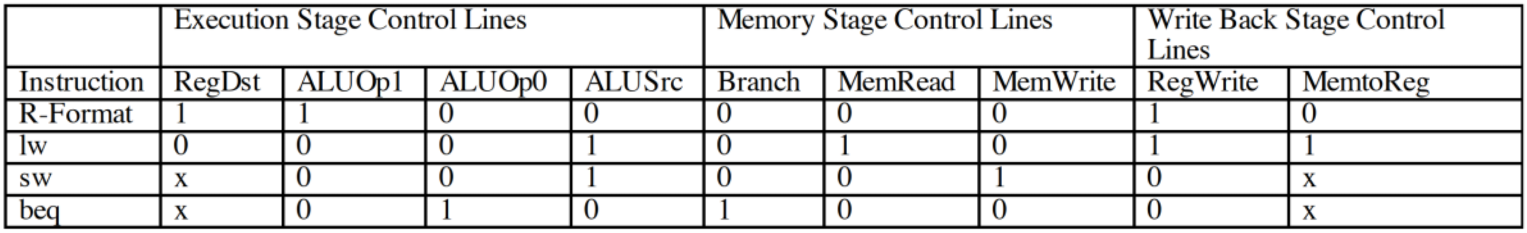
Tuesday, April 19, 2022 3:55 PM

Idea: the control signals for a pipeline processor are the same as the single cycle processor. They can be generated once, then use DFFs to propagate control signals as they follow their instruction. <u>Control flows through the pipeline along with data.</u>

A possible design for the Pipelined Machine is:



The control signals are identical, just split into stages:

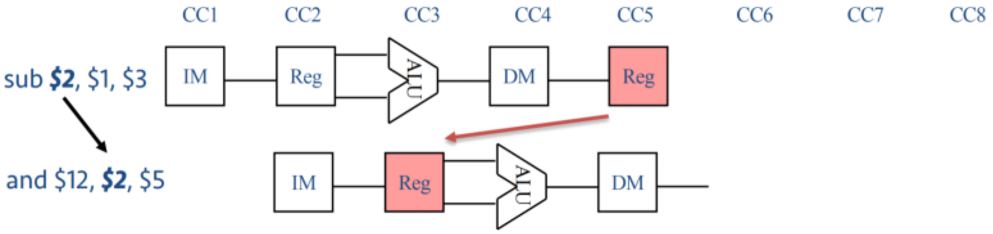


	Execution Stage Control Lines				Memory Stage Control Lines			Write Back Stage Control Lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-Format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	x	0	0	1	0	0	1	0	x
beq	x	0	1	0	1	0	0	0	x

Data Hazards

Thursday, April 21, 2022 3:37 PM

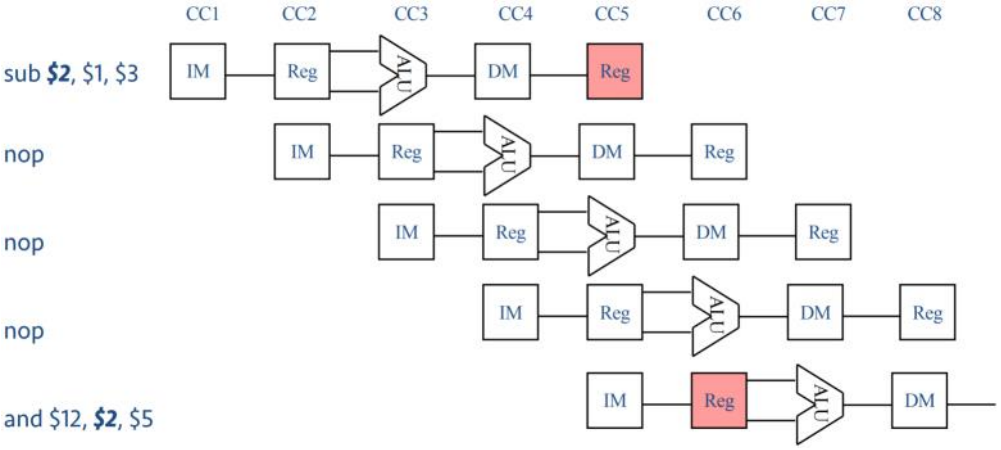
Problem: The next instruction in the pipeline may depend on a writeback from the previous instruction!
Example:



Software Solutions:

- Use nop instruction (add $0 $0 $0)

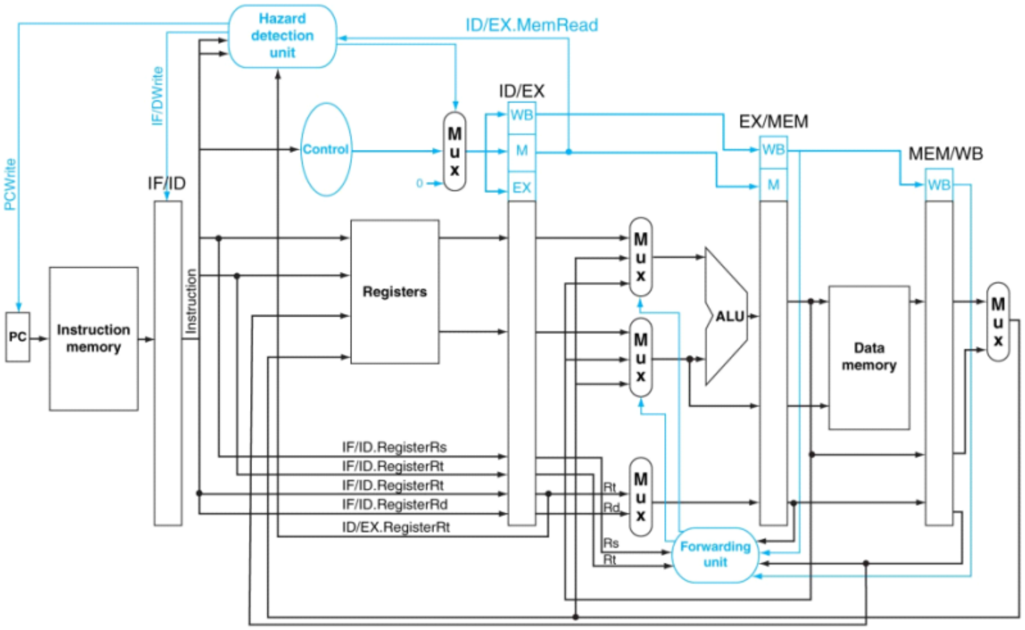
Nop Example:



Hardware Solutions:

- Stalling the pipeline until first instruction has resolved
- Forwarding, send result of ALU back to register file before the first instruction completes

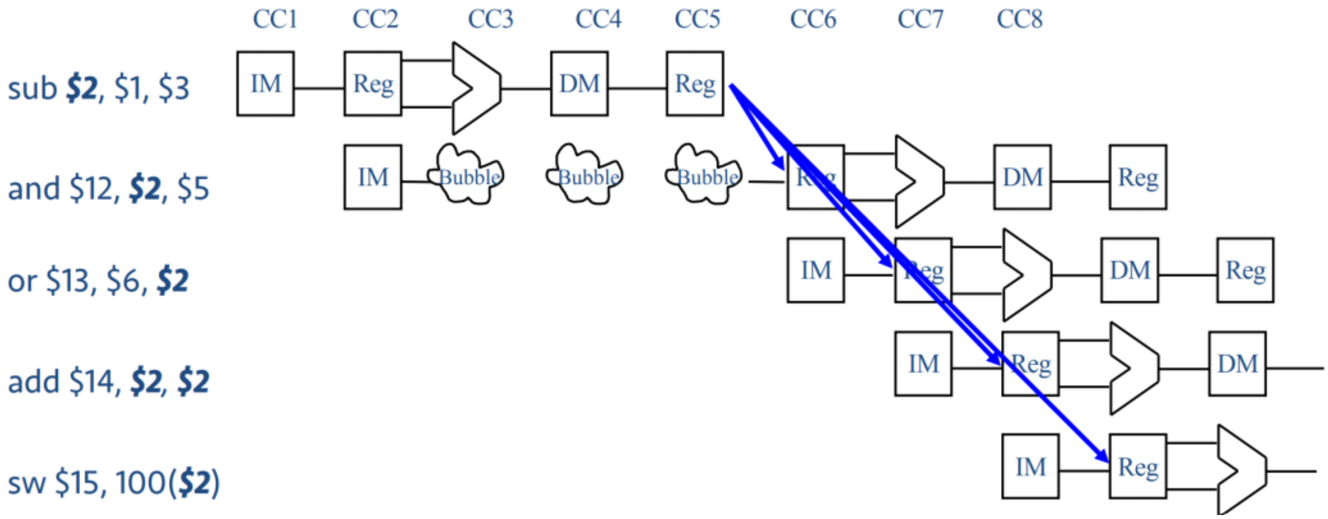
A possible implementation of stalling and forwarding might look like:


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
0
M u x
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Instruction
PC
Instruction memory
Registers
M u x
M u x
ALU
Data memory
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Rt
Rd
M u x
Rs
Rt
Forwarding unit

Stalling

Thursday, April 21, 2022 4:00 PM

Idea: We can insert nops in the pipeline to resolve Data Hazards
Stalling Example:



Note that stalling the second instruction resolves the future data hazards

Implementation:

- Set control signals to ID/EX Registers to 0 (send a new nop instruction)
- Set PCWrite to 0 (don't increment)
- Set IF/ID Register write to 0 (keep trying to decode the next instruction until it is safe to run)

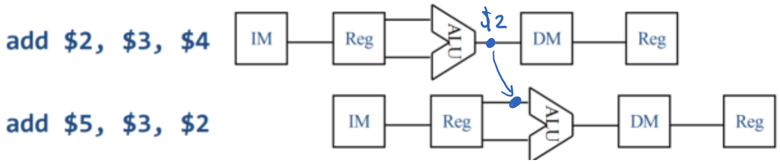
The register write address and RegWrite signals are stored through the stages of the pipeline. Use these signals to determine whether to stall.

Stalls should occur after fetch but before decode.

Forwarding

Thursday, April 21, 2022 4:33 PM

Idea: Forward the ALU's result ahead of time.
Forwarding Example:



```
add $2, $3, $4
add $5, $3, $2
```

Can handle EX hazards, MEM hazards, and also WB hazards

Cannot handle every hazard with forwarding because we want to only forward values in registers

At the end of Execute and Memory stages, we can send the result to the beginning of another instruction's Decode or Execute stage

Branch Hazards

Tuesday, April 26, 2022 3:59 PM

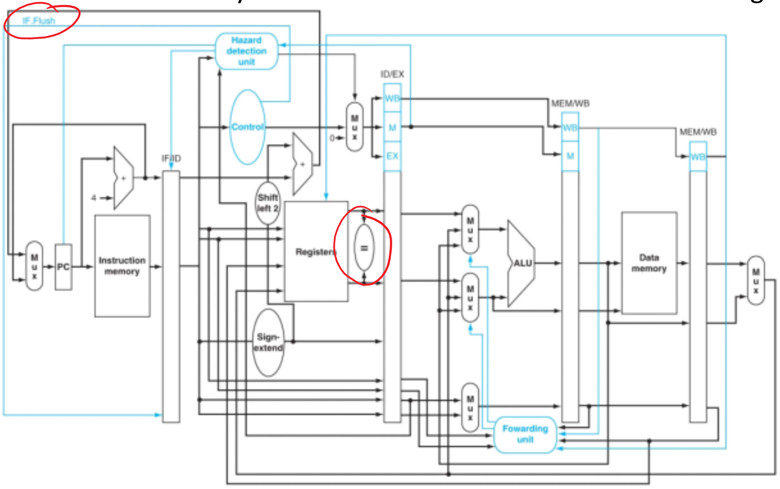
Idea: We may execute code after branches before resolving the branch outcome

Solution:

- Use stalling: stall the pipeline until the branch decision is resolved
- Guess: keep doing the instructions until the decision is resolved

Implementation:

- Branch Target Buffer: keeps track of addresses which are branches, allows us to tell when a branch will be fetched
- Reduce Branch delay: move branch outcome to the decode stage by adding comparator to register file outcome



- Branch delay slot: instruction after the branch will always executed even if the branch is taken
  Fill branch delay slot with:
  - Instructions before the branch (must not violate dependencies)
  - Instructions after the branch which will be overridden if the branch is taken
  - Instructions after the branch target which will be overridden if the branch is not taken
- Branch prediction: try to guess which path will be taken

In practice, modern machines have large branch penalties and therefore would have huge branch delay slots which is not ideal

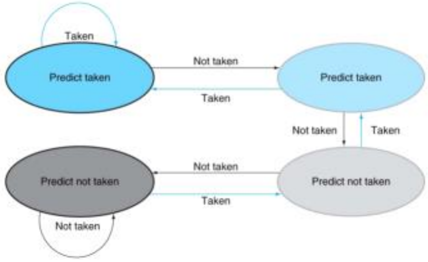
Branch Prediction

Thursday, April 28, 2022 3:33 PM

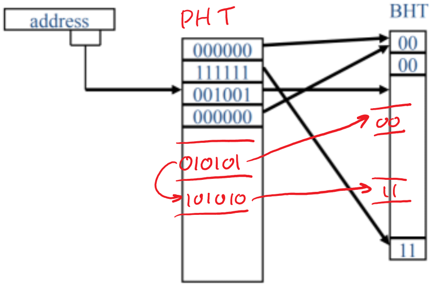
Problem: Predicting always taken or including branch delay slots is not useful when the pipeline becomes large

Solution: Modern branch prediction strategies:

- Static predictors: for branch B, always take the same branch
- Dynamic predictors: for branch B, make a new prediction every time the branch is called
    - What did the branch take previously? Keep table of previous predictions
    - 1 bit predictor: keep table of 1 previous branch result, predict the same result in the future
    - 2 bit predictor: keep table of 2 previous branch results, used state machine shown below to predict



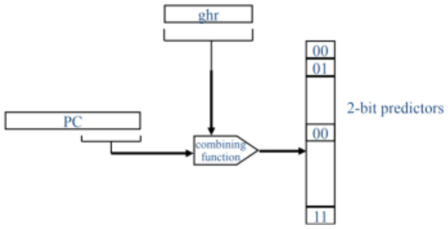
    - 2 level local predictor: store a pattern of branch histories and then use pattern as address for another predictor



Example: TN TN TN ....

Operations: Find if branch
Lookup branch in PHT
Use PHT as address in BHT, make prediction
Update BHT using actual result
Update PHT using actual result

    - Global history register: keep branch pattern for all branches that have run and feed into predictor



    - Combining branch predictors: use multiple branch predictors and have a chooser to pick the best predictor

Tradeoffs? Static predictors are easier to implement but dynamic predictors are more accurate

Aliasing

Thursday, April 28, 2022 4:41 PM

Idea: Because pattern history tables, branch predictor state tables, etc have limited size: then multiple branches may overlap in the table, creating aliasing

The Standard Machine

Thursday, April 28, 2022 4:45 PM

The Standard Machine has:

- 5 pipeline stages
- Stalls and forwarding enabled
- Early branch resolution (branch outcome computed in ID stage), one branch prediction slot
- Some sort of branch prediction

Advanced Pipelining

Tuesday, May 10, 2022 3:24 PM

Jump Predictors, Jump History Table

Tuesday, May 10, 2022 3:31 PM

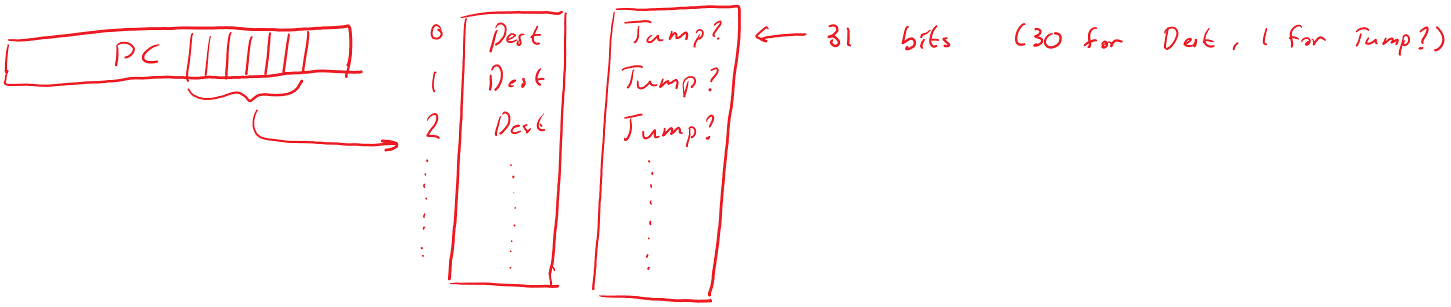
Jumps: we can use a BDS to avoid stalls or flushes

- Jumps resolved in ID stage can have 0 stalls/flushes with a BDS

Problem: We want to eliminate the flush for jumps

Solution: If IF stage can remember it is a jump, we can jump immediately before the next cycle

- Use a table to store the PC values for jumps: Jump History Table (JHT)
- Can be used for both J and Jr



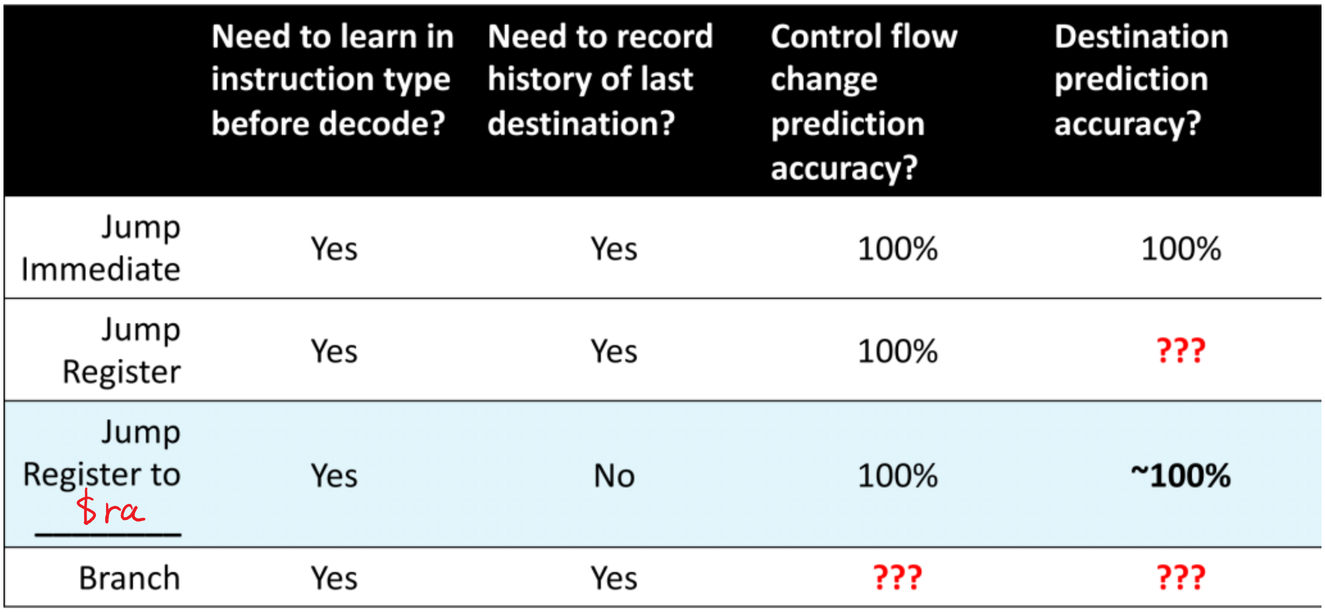
← 31 bits (30 for Dest, 1 for Jump?)

- Predict that the jump will go to the same destination every time

Branch/Jump Predictor Data Structures

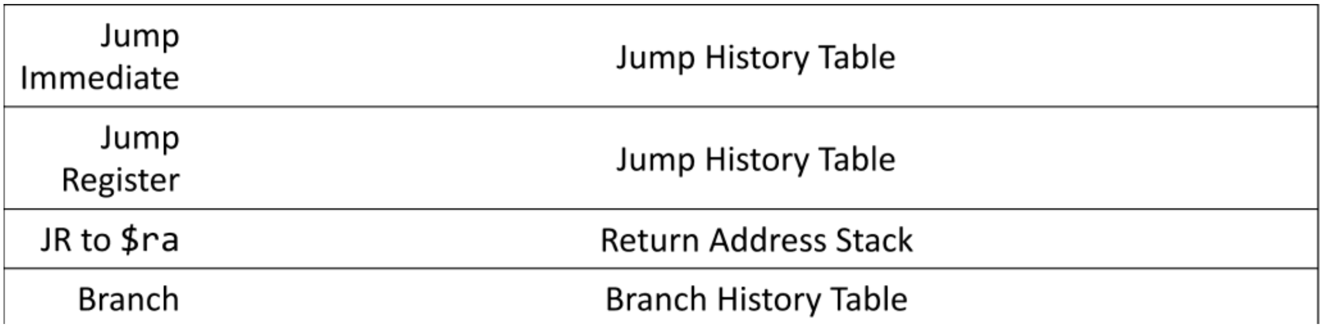
Tuesday, May 10, 2022 4:02 PM

Summary of jump and branch information requirements and prediction accuracy



	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Yes	Yes	100%	100%
Jump Register	Yes	Yes	100%	???
Jump Register to $ra	Yes	No	100%	**~100%**
Branch	Yes	Yes	???	???

Data structures used to store information for each



Jump Immediate	Jump History Table
Jump Register	Jump History Table
JR to $ra	Return Address Stack
Branch	Branch History Table

Return Address Stack: Is the instruction a return? Where to return?

- Create table to store whether instruction is a return.
- Create a stack with return addresses.

Exceptions and Interrupts

Thursday, May 12, 2022 3:50 PM

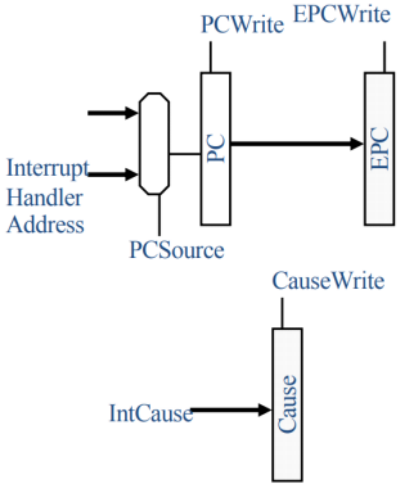
Def: Exceptions are another type of non-sequential control flow

- Exceptions are typically asynchronous and non-deterministic
- Any unexpected change in control flow

Def: Interrupts are any externally caused exceptions

Types of exceptions:

- Arithmetic exceptions
- Illegal memory access
- Illegal instruction



Idea: On exception we need to

- Save the PC
- Record nature of exception or interrupt
- Transfer control to OS

Handling Exceptions:

- Add exception PC, which holds PC value of exception
- Add exception cause register, hold information about exception
- Controls to write to exception PC and exception cause PC

Note: Exceptions must be caught early in the pipeline to avoid any permanent changes to state from later instructions

- For the standard machine, the latest exception must be raised during the memory stage

Modern Processors

Thursday, May 12, 2022 4:17 PM

Idea: most modern processors have deeper pipelines and:

- Superscalar execution

  Idea: use many copies of the same processor and work on multiple instructions in parallel

  Limitation: can only work on independent instructions in parallel because not all components (memory) can be parallel
- Out-of-order execution

  Idea: find multiple instructions which are independent and execute them out of order

  Limitations: Difficult to build
- Very-large-instruction-word

  Idea: make instruction words encode multiple tasks in one word, push solving parallelism to the compiler

  Limitation: relies on the performance of the compilers

Note: a N-issue superscalar processor fetches N instructions at the same time

- Dynamic Scheduling or Out-of-order scheduling

  Idea: Begin execution of instruction as soon as all of its dependencies are satisfied
- Register Renaming:

  Idea: make more physical registers than used by the compiler, avoids write after write hazards

Realistic Memory and Caches

Tuesday, May 17, 2022 3:26 PM

Note: made an assumption that memory can be accessed in 1 cycle, which is generally true for lower power processors

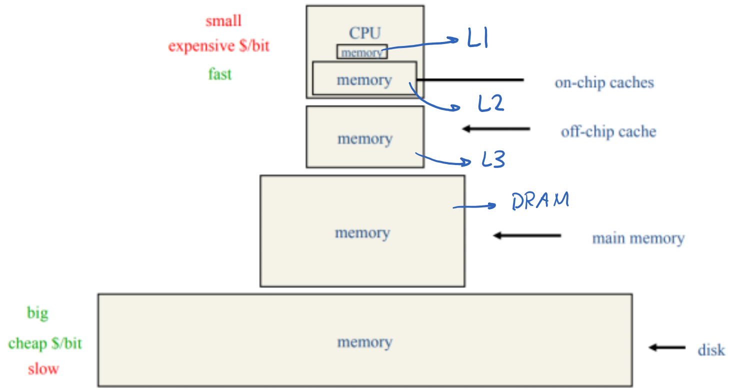
Problem: Faster processors may take hundreds of cycles to access memory
Solution: Store important data closer to the processor core in a cache structure

- Should be a small structure on order of KB to reduce delay
- Should be close to the processor to reduce latency

Locality:

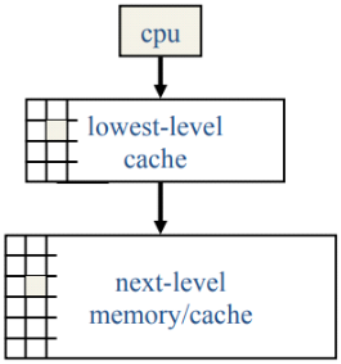
- Store data which is close in space or time close to the processor
- near in time: we will often access the same data again very soon
- near in space: our next access is often very close to our last access

Cache uses a tiered structure to manage locality



Cache Fundamentals

Thursday, May 19, 2022 3:37 PM



Cache hit: Access where the data is found in the cache
Cache miss: An access where the data is not in the cache
Hit time: Time to access the cache
Miss penalty: time to move missed data from further level to closer
Hit ratio: percentage of the time data is found in the cache
Miss ratio: (1 - hit ratio)

Cache block/line size: amount of data that gets transferred on a cache miss

- Implicitly supports spacial locality

Instruction cache: holds instructions
Data cache: holds data
Unified cache: golds both instructions and data

Problems to consider:

- On memory access:
  - how to know if it is a hit or miss
- On cache miss:
  - Where to put new data?
  - What data to throw out?
  - How to remember what data was saved?

Implementation:

- Size of cache is inversely proportional to speed. Smaller = faster
- Caches will store minimal needed bits to work

Fully Associative Cache

Thursday, May 19, 2022 3:50 PM

Where to put? Anywhere
What to replace? Least recently used

address string:

4	00000100	Miss
8	00001000	Miss
12	00001100	Miss
4	00000100	Hit
8	00001000	Hit
20	00010100	Miss
4	00000100	Hit
8	00001000	Hit
20	00010100	Hit
24	00011000	Miss
12	00001100	Miss
8	00001000	Hit
4	00000100	Miss

the *tag* identifies the address of the cached data

tag	data
000011 ~~000001~~	m[12:15] ~~m[4:7]~~
00001	m[8:11]
000001 ~~000101~~	m[4:7] ~~m[20:23]~~
000110 ~~000011~~	m[24:23] ~~m[12:15]~~

4 entries, each block holds one word, any block can hold any word.

Direct Mapped Cache

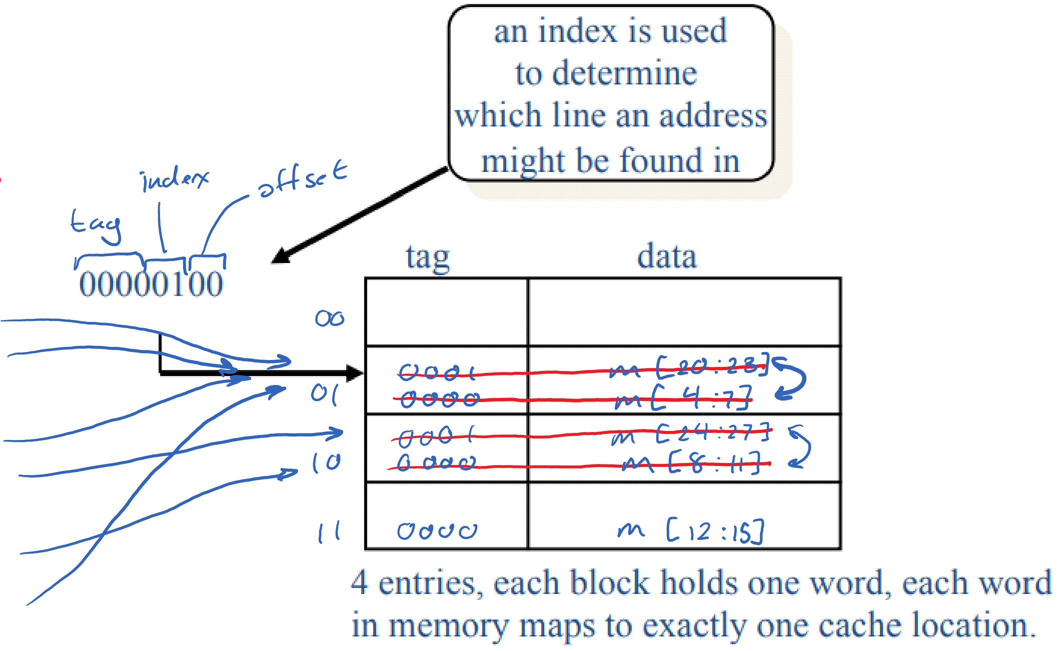
Thursday, May 19, 2022 3:53 PM

Where to put? A specific spot

What to replace? Whatever is in the spot

address string:

4	00000100	Miss
8	00001000	Miss
12	00001100	Miss
4	00000100	Hit
8	00001000	Hit
20	00010100	Miss
4	00000100	Miss
8	00001000	Hit
20	00010100	Miss
24	00011000	Miss
12	00001100	Hit
8	00001000	Miss
4	00000100	Miss



an index is used to determine which line an address might be found in
tag
index
offset
00000100

	tag	data
00		
01	~~0001~~ ~~0000~~	~~m[20:23]~~ ~~m[4:7]~~
10	~~0001~~ ~~0000~~	~~m[24:27]~~ ~~m[8:11]~~
11	0000	m[12:15]

4 entries, each block holds one word, each word in memory maps to exactly one cache location.

Note: Uses specific bits to determine row that the tag (remaining bits) is placed

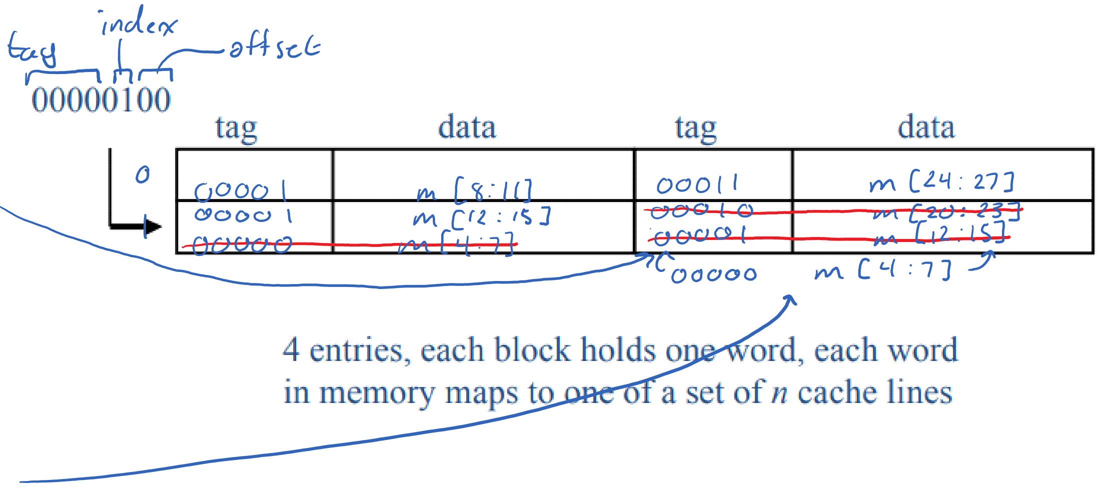
N-way Set Associative Cache

Thursday, May 19, 2022 4:15 PM

Where to put? A specific row and some column

What to replace? Something from that row which was least recently accessed

address string:		
4	00000100	Miss
8	00001000	Miss
12	00001100	Miss
4	00000100	Hit
8	00001000	Hit
20	00010100	Miss
4	00000100	Hit
8	00001000	Hit
20	00010100	Hit
24	00011000	Miss
12	00001100	Miss
8	00001000	Hit
4	00000100	Miss



tag	data	tag	data
00001	m[8:11]	00011	m[24:27]
00001	m[12:15]	~~00010~~	~~m[20:23]~~
~~00000~~	~~m[4:7]~~	~~00001~~	~~m[12:15]~~
		00000	m[4:7]

4 entries, each block holds one word, each word in memory maps to one of a set of *n* cache lines

Associativity = number of blocks per set

Benefit of associative caches:

- Higher hit rate

Detriment of associative caches:

- Slightly larger (longer tags)
- Slightly slower (need to do a linear search)

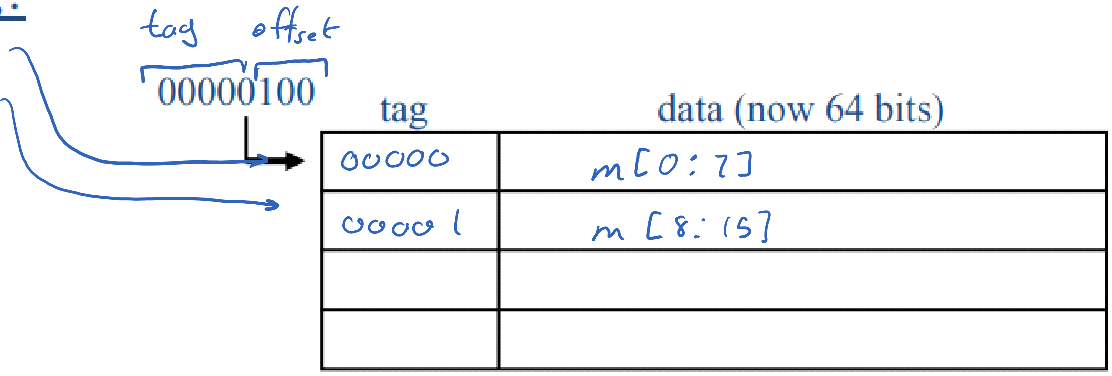
Larger Cache Blocks

Thursday, May 19, 2022 4:26 PM

Idea: Store more words per cache block entry

address string:

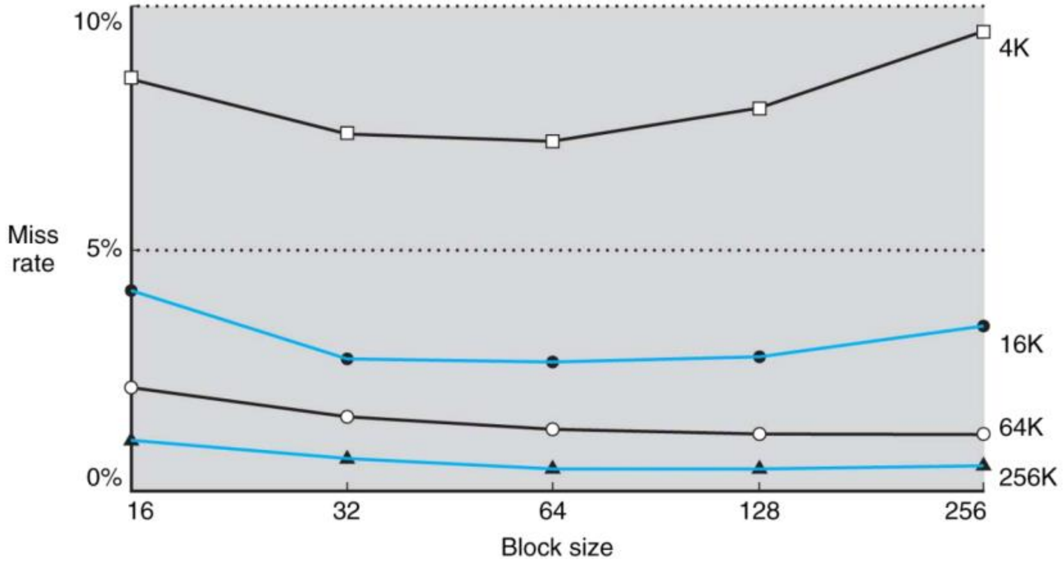
4	00000100
8	00001000
12	00001100
4	00000100
8	00001000
20	00010100
4	00000100
8	00001000
20	00010100
24	00011000
12	00001100
8	00001000
4	00000100



tag	data (now 64 bits)
00000	m[0:7]
00001	m[8:15]

4 entries, each block holds two words, each word in memory maps to exactly one cache location (this cache is twice the total size of the prior caches).

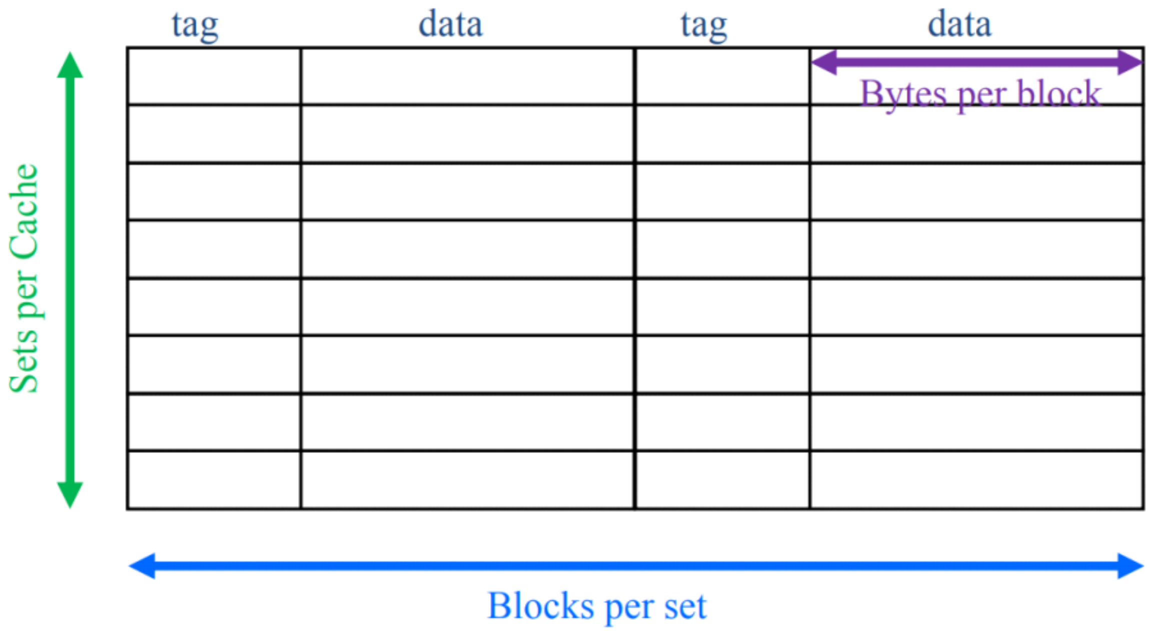
But: There are diminishing returns for larger cache sizes



Cache Parameters, Cache Performance

Thursday, May 19, 2022 4:36 PM

Cache size = Number of sets * block size * associativity



Note: Cache size only considers cache data, not metadata (tags, valid bits)

Cache Performance:

CPI = BCPI + MCPI

BCPI = base CPI assuming perfect memory

MCPI = miss CPI, number of cycles per instruction when a miss occurs

Handling Cache Accesses, Cache Alignment

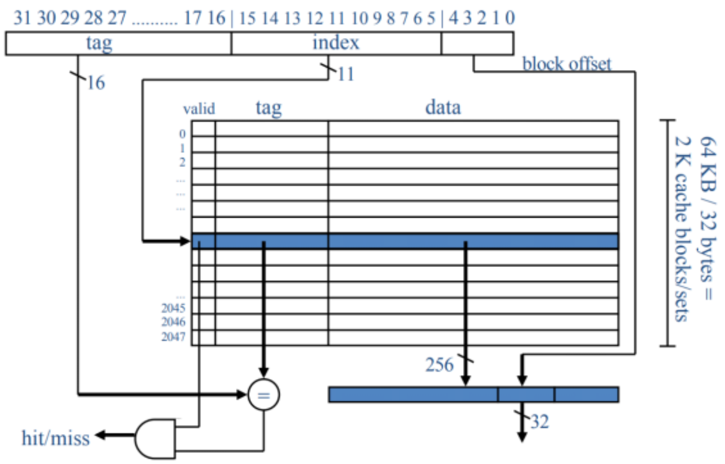
Tuesday, May 24, 2022 3:37 PM

Procedure for accessing memory location:

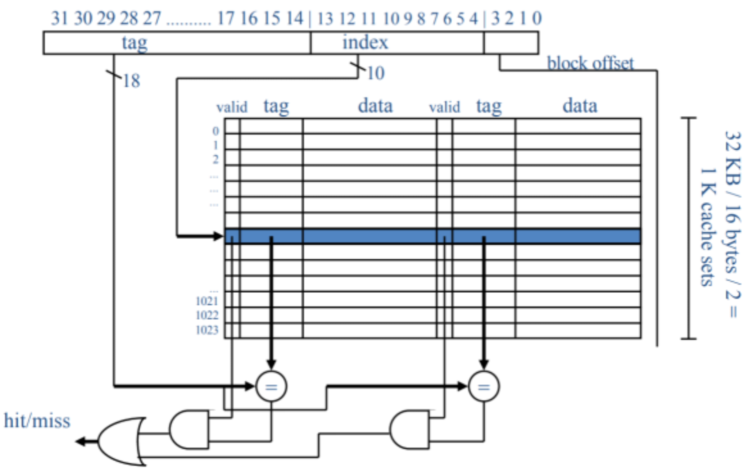
1. Use index and tag to access cache and determine hit/miss.
2. If hit, return requested data.
3. If miss, select a cache block to be replaced, and access memory or next lower cache (possibly stalling the processor).
   - load entire missed cache line into cache
   - return requested data to CPU (or higher cache)
4. If next lower memory is a cache, goto step 1 for that cache.

Example:

64 KB cache, direct-mapped, 32-byte cache block size



32 KB cache, 2-way set-associative, 16-byte block size

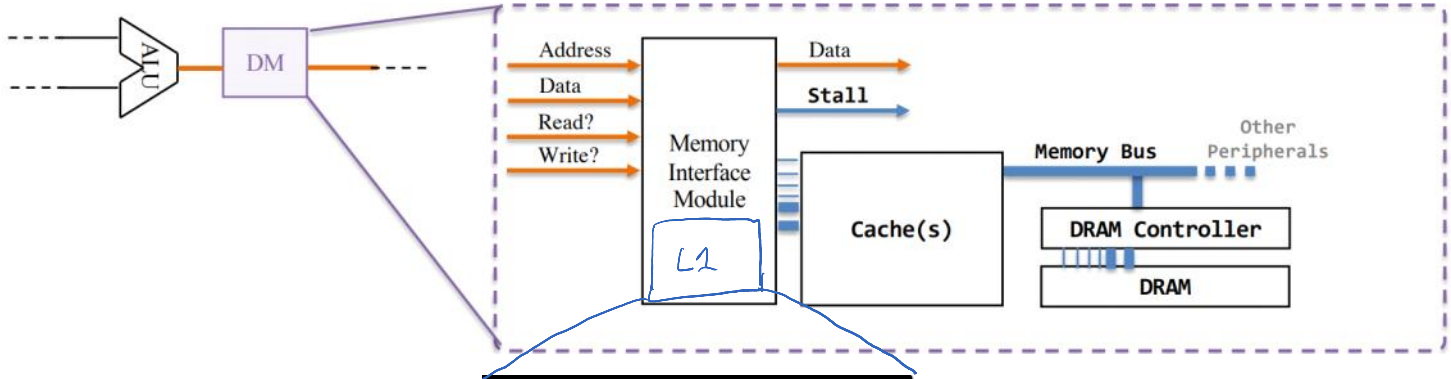


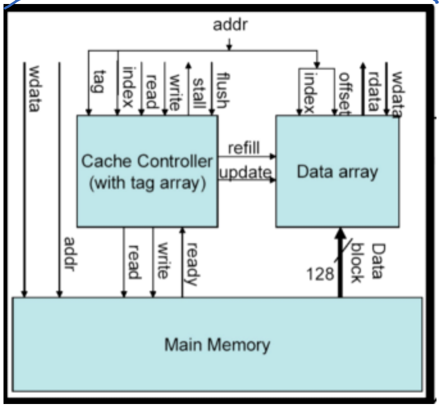
Cache Alignment: Data which is loaded into the cache must be data which shares the same index and tag

Connecting Cache to the Pipeline

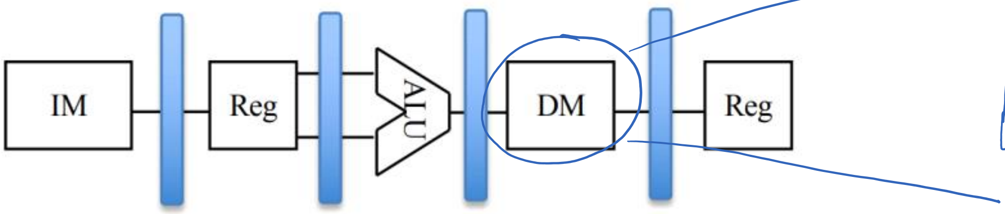
Tuesday, May 24, 2022 3:50 PM

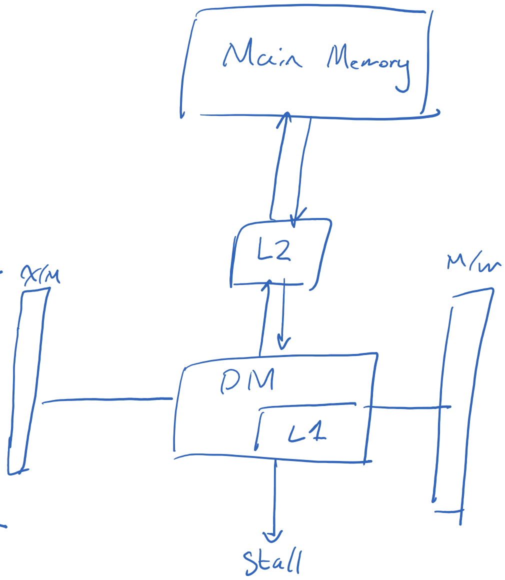
A simplified view of the Data Memory module expansion with cache





In the pipeline:





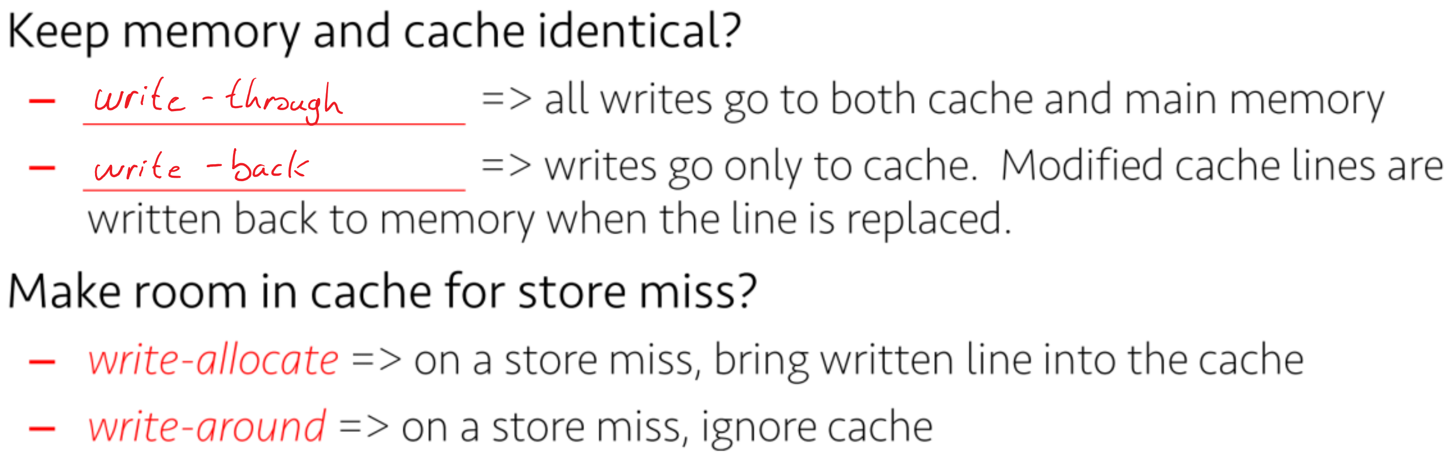
Cache Stores

Tuesday, May 24, 2022 4:26 PM

Problem: Stores don't necessarily stall the CPU, but they change contents of memory

- Need to guarantee the caches and memory are all synced together when a store is executed
- Empty cache might need to pull from memory before storing a word in the cache block

Solution: policy decisions for stores



Keep memory and cache identical?

- write - through => all writes go to both cache and main memory
- write - back => writes go only to cache. Modified cache lines are written back to memory when the line is replaced.

Make room in cache for store miss?

- *write-allocate* => on a store miss, bring written line into the cache
- *write-around* => on a store miss, ignore cache

Types of Cache Miss, Solutions to Misses

Tuesday, May 24, 2022 4:39 PM

Compulsory miss: first time access to data
Capacity miss: Missed only because the cache isn't large enough
Conflict miss: Missed because data maps to same line as other data and was forced out

Solutions:
Compulsory misses: larger block sizes to load more initial values
Capacity misses: more capacity
Conflict misses: more associativity

Advance Cache Architectures, Handling Misses

Thursday, May 26, 2022 3:29 PM

Terms:

Average Memory Access Time (AMAT) = hit time + miss rate * miss penalty

Ways to improve AMAT:

- Decrease hit time
- Decrease miss rate
- Decrease (observed) miss penalty

Idea: Victim cache stores recently evicted cache entries

- Alleviates conflict misses

Idea: Prefetching accesses memory before core needs it

- Can be done in hardware
- Can be done in software after profiling program for possible speedups
- Separate thread to prefetch data for another (speculative precomputation)

Reducing memory stalls:

- Non-blocking cache: cache that can handle new requests after a miss
- Hit-under-miss: can service hits after one miss, stalls on second miss
- Miss-under-miss: can have many outstanding misses before a stall

Tolerating cache misses:

- Stall on miss (no tolerance)
- Stall on use (keep doing other instructions until miss is used)
- Non-blocking caches (service other requests after a miss)
- Out-of-order execution (execute other instructions out of order)
- Multithreaded execution (run multiple processes while waiting for memory)

Virtual Memory

Thursday, May 26, 2022 3:52 PM

Problem: what happens if two programs in the processor uses the same memory addresses? What happens if a program accesses memory that doesn't exist?

Idea: Abstract physical memory into virtual space

- To program, looks like all of memory is available
- Maps virtual addresses to actual physical memory, uses disk or larger memory to handle overflows

Def: another level in cache/memory hierarchy

- Allows use of large memory space (on disk)

Terminology:

cache	VM
block	page
cache miss	page fault
address	virtual address
index	physical address (sort of)

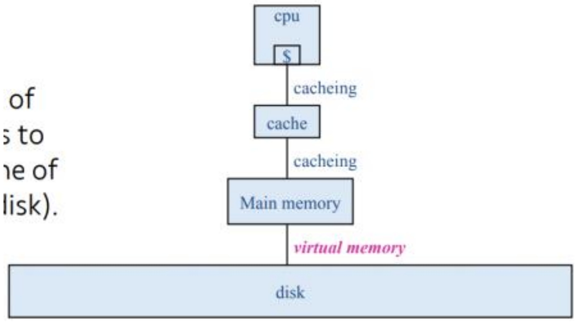
Difference from memory caches:

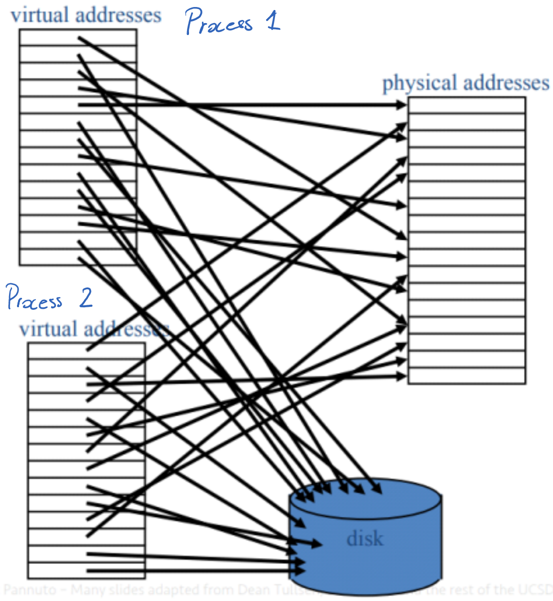
- Miss penalty of millions of cycles

Design Decisions:

- Large pages (4KB to MB)
- Associative mapping (usually fully associative)
- Software handling of misses but not hits
- Write-back only

of
s to
ne of
lisk).





Pannuto – Many slides adapted from Dean Tullsen ... the rest of the UCSD f

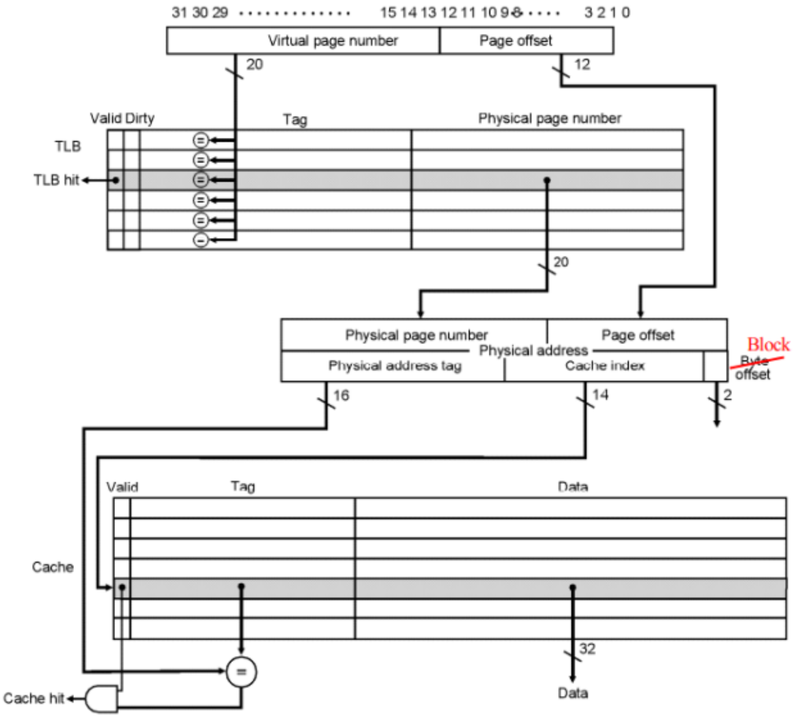
Virtual Address Translation

Thursday, May 26, 2022 4:03 PM

Translate addresses by changing only a few bits of the address

- Only translate the high order bits
- Leave lower bits the same, defines the page size (analogous to block offset)

TLB: Translation Lookaside Buffer stores some virtual page data



Note: Cache lookup is now a serial process

- V->P translation through TLB
- Get index
- Read tag from cache
- Compare

Improvements:

- If V->P does not change the index, then they can be done in parallel, so index needs to come from only the page offset