CS 154 Notes: Computer Architecture

Classes of Computers:

- Personal Computers: general purpose, variety of software [Constraints: costs, battery, size]
- Servers: the cloud; high performance, capacity, and reliability [Constraints: low cost]
- Supercomputers: high-end scientific and engineering calculations [Constraints: highest performance]
- Embedded Computers: hidden as components of systems [Constraints: tiny, zero power, minimal cost]

Applications suggest how to improve technology and provide revenue to fund development

Improved technologies make new applications possible

Cost of software development makes compatibility a major decision market in the market

Computer Components: (1) Processor, (2) Memory, (3) Input Devices, (4) Output Devices, (5) Data Storage

Motherboard Contents: IO Devices, CPU Chip, Graphics Chip, Power Connectors, Memory IO Connectors

Parts of the CPU:

- **Datapath:** ALU, registers, functional units that data moves through to perform calculations
- **Cache Memory:** small, fast memory inside the CPU with frequently used data
- (Main memory lives outside the CPU on a separate DRAM chip)
- **Control Logic:** sequencing how datapath and memory interact

Fetch-Execute Cycle:

- **Instruction Fetch (IF):** fetch the next instruction in the program (from memory)
- **Instruction Decode (ID):** the instruction (in control unit)
- **Execution (EX):** Execute the instruction (in the alu)
- **Memory (MEM):** access stored data as necessary and read/write (in memory)
- **Write Back (WB):** Write Back results to registers (in registers)

Pipelining in CPUs: pipelines can make CPUs faster (but need to prevent data hazards)

- **Instruction Level Parallelism:** allow multiple instructions to go on at once

Computer Languages and the FE Cycle:

- CPU executes machine language instructions, which need multiple cycles to run

Machine and Assembly Language:

- CPU executes machine level instructions
- Compilers/Interpreters translate higher level languages to lower level languages
- Assembers translate lower level languages into machine language

Computer Memory: <Address: location of data, Data: the stored data>

- DRAM: dynamic RAM (prices go down with improvement in capacity)

Abstraction Layers:

Application
Algorithm
Programming Language
Operating System
Hypervisor
Instruction Set Architecture
Microarchitecture
Register-Transfer Level
Gates
Circuits
Devices
Physics

Manufacturing Integrated Circuits (ICs): (1) prepare wafer, (2) apply photoresist, (3) align photomask, (4) expose to UV light, (5) develop & remove exposed photoresist, (6) etch exposed oxide, (7) remove photoresist

Cost of Manufacturing ICs:

- Cost per die = Cost per wafer / (Dies per wafer * yield)
- Dies per wafer ~= Wafer area / Die area (round down)
- Wafer cost and Die area are usually fixed values (area determined by design)
- Yield = good die / total die = 1 / [1 + (Defects per area * Die Area / 2)]^2

Metrics:

- **Latency:** response time to complete a single fixed task (calculations emphasizes)
- **Throughput:** total work done in a fixed time (server emphasizes)

Performance Measures:

- **Execution Time:** total response time (CPU + External Memory I/O + OS + idle), which determines overall system performance
- **CPU Time:** time spent processing a given job (excludes I/O, OS times)
    - determined by program length, computer architecture, CPU clock speed
    - can improve performance by reducing number of clock cycles and increasing clock rate
    - has tradeoff between clock rate against cycle count
    - not all instructions use the same number of clock cycles
- Performance = 1 / Execution Time
- Relative performance (Speedup) of system A vs B = P_A / P_B
- Speedup Latency = Latency Old / Latency New
- Speedup Throughput = Throughput New / Throughput Old
- CPU Time = CPU Clock Cycles * Clock Cycle Time = CPU Clock Cycles / Clock Rate
- CPU Time = instructions/program * clock cycles/instruction * seconds/clock cycle
- CPU Time = instruction count * CPI * cycle time

CPU Clocking: hardware operates on a constant-rate clock

- **Clock Period:** duration of a clock cycle
- **Clock Frequency (Rate):** cycles per second
- Clock Frequency = 1 / Clock Period

Challenge: Power Consumption: market wants power consumption to decrease while increasing performance

- Power = Capacitive Load * Voltage^2 * Clock Frequency
- Contributes to Moore's Law Plateau

Challenge: Idle Power: CPUs draw disproportionate power when idling

Challenge: Multiple Processors: more than 1 processor per chip

- Requires explicit instruction level parallel programming (hidden from programmer)
- Need to maximize performance, balance the load, optimizing communication and synchronization

Challenge: Amdahl's Law: improve an aspect of a computer and expect a proportional overall improvement

- $T_{improved} = T_{unaffected} + T_{affected}$ / improvement factor

Pitfall: MIPS as a Performance Metric: MIPS = Millions of Instructions per second

- Doesn't account for ISAs between computers (which have different efficiencies)
- Differences in complexities between instructions (weighted CPIs)
- Better to stick to CPU Time per Fixed Process

Multiplexer: M bits N inputs to 1 output

- 2:1 Mux: if S == 0, return A, else if S == 1, return B

Transpile: translating source code into another language with similar level of extraction

Hardware Description Language (HDL): used for simulation

Instruction Set Architectures (ISAs): abstract contract between software and hardware

- described with programmer-visible states, semantics/syntax and instructions
- can have many different implementations, some of which are customizable

Classification of ISAs:

- By complexity: Complex Instruction Set Computer (CISC) vs Reduced Instruction Set Computer (RISC)
    - instruction complexity (CPI), transistors, power, commercial vs embedded
    - RISC: simply instructions
    - CISC: as few lines of code as possible
- By parallelism: Very Long Instruction Word (VLIW) vs Explicit Parallel Instruction Computing (EPIC)
    - less commercial, server/supercomputer usage
- By simplification: Minimal Instruction Set Computer (MISC) vs One Instruction Set Computer (OISC)
    - proof of concept, little parallelism

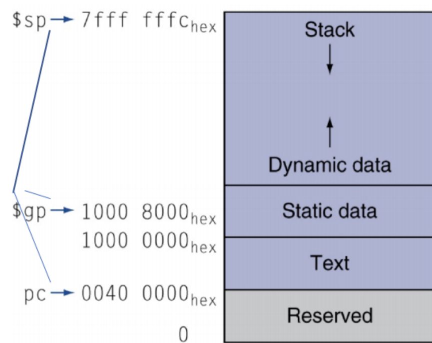
MIPS ISA: instructions in either R (register), I (immediate), or J (jumping) format represented with 32 bits

- for immediate instructions, the upper 16 bits is set to 0
- Pseudo Instructions: not core to CPU and are slower to run than core instructions

Variables: have type and storage class

- **Automatic variables:** local to part of program; created and discarded
- **Static variables:** global variables
- MIPS uses the global pointer register $gp to access static variable block

Memory Layout:



- **Text:** program code
- **Static data:** global variables
- **Heap:** dynamic data (created and destroyed by programmer)
    - Heap is in upward direction (bottom of heap is lower memory address)
    - Stack and heap grow toward each other
- **Stack:** automatic storage of local variables (created and destroyed by compiler)
    - Stores arguments and variables when functions are called
    - Stores large local variables that don't fit in registers
    - Stack is in reverse direction (high memory is bottom of stack)

Character Data: byte-encoded character sets

- **ASCII:** 8 bits, includes all english characters but not non-English characters
- **Unicode** (-8, -16, -32): 8/16/32 minimum number of bits for variable character encoding
- Must be stored in memory (.data directive) and load them from memory (must fill to be 32 bits)

String Representations:

- 1st Choice: give length of string in the first position
- 2nd Choice: accompanying variable for length of string
- 3rd Choice: terminating EOS \0 character (C-Strings)

Branch Addressing: 16 bit immediate in WORDS

- Immediate types, can add or subtract done relative to PC Register
- Most branch targets are near branch instruction in text segment of memory
- Target Address = PC + 4 * Word Offset
    - Note: PC is automatically incremented by 4 when this calculation is done
- Word alignment allows us to jump further with the 16 bit immediate restriction
- If 16 bit offset is too small to encode far branch target, use jump addressing with 26 bit limit

Jump Addressing: 26 bit address in WORDS

- Encode full absolute address in instruction, not relative addressing
- Target Address = 4 * Address (in Words) | PC[31:28]
    - Get 4 most significant digits from PC (If need to cross boundary, jump and add and jump)
    - Concatenate 26 bits in address; Concatenate two zeroes

Parallelism and Synchronization using Instructions

- Data Race is processors access same shared area of memory
- Need hardware support for lock and unlock
- Atomic memory operation = no other memory access allowed until atomic operation finished
- Load link (ll) and Store conditional (sc) used in sequence
    - Load link make reservation on a certain register
    - Store conditional attempts to store at register; returns value in rt (0 = fail, 1 = success)

File to Machine Code

1. **Compiler:** HLL to Assembly; may have assemblers and linkers built-in
2. **Assembler:** take care of pseudo instructions and number conversions (to hex); produces object file (machine language instruction, data, and information needed to place instructions properly in memory)
    - must determine addresses of labels and resolve labels for branching/jumping instructions
3. **Linker:** combine all object files into executable program, resolving symbols (references) as it goes along; products a single executable file with machine language instructions
4. **Loader:** OS program that takes executable code, sets up CPU memory for it, copies over the instructions to CPU memory, initializes all registers, jumps to the start routine)

Note: since a pointer is a memory address, when we simply add 4 to the address, we don't need additional instructions for indexing from an array; compilers can optimize code for this.

Dynamic Linking: only finish linking a library procedure when it is called

- PRO: often used libraries only need to be stored in a single location and not duplicated
- PRO: Update/fixes to library can be done modularly
- PRO: Reduce compiling time
- CON: Newer version of library is not backward compatible

Java: compiled to Java bytecode instruction set; usually slower than C/C++

- Run on Java virtual machine regardless of computer architecture
- JVM is a software interpreter that simulates an ISA
- PRO: portability
- Performance can be enhanced with Just in Time compilation

Addition and Subtraction Overflow:

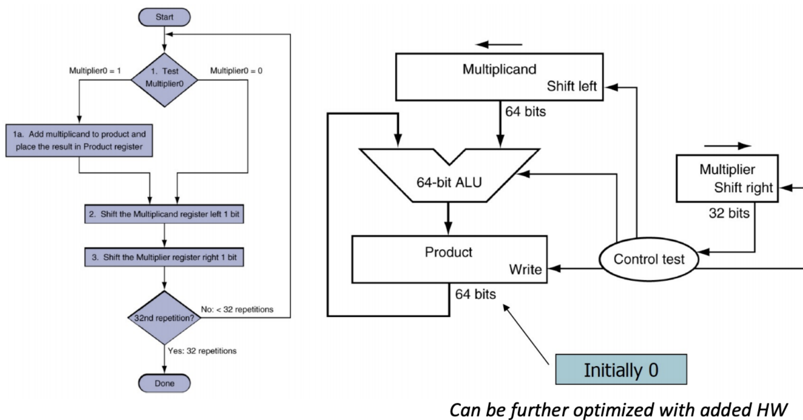
- Languages like C/C++, Java ignore overflow. Use unsigned assembly instructions to simulate this.
- Demand on CPU runtime is excessive; we can manually check in such languages.
- When using the signed operations, an exception handler is invoked:
    - PC is saved in exception program counter register
    - Jump executed to a predefined handler address

Multiplication and Division:

- mult multiplies 2 numbers and stores lower 32-bit result in LO and higher 32-bit result in HI
- div divides 2 numbers and stores the result in LO and quotient in HI
- use mflo and mfhi to move out of lo and hi registers
- no checking for overflow or divide by zero

Multiplication Algorithm: for M * N

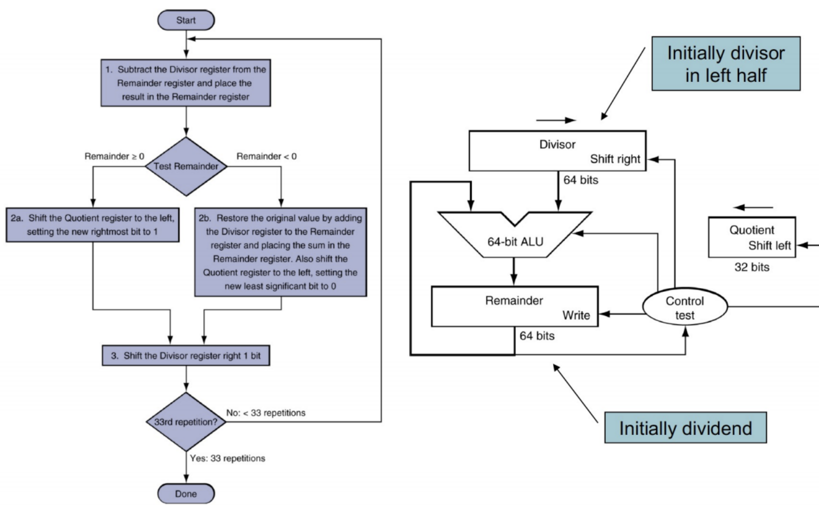
- Initialize P = 0
- Loop 32 times:
  - If bit 0 of N is 1, P += M
  - Shift N to the right and M to the left 1 bit
- Return P (product)



Can be further optimized with added HW

Division Algorithm: for N / D

- Initialize R = N
- Loop 32 times:
  - R = R - D
  - If R >= 0, shift Q to the left one bit and set last signed bit to be 1
  - Else, R = R + D and shift Q to the left one bit
  - Shift D to the right 1 bit
- Return R (remainder) and Q (quotient)



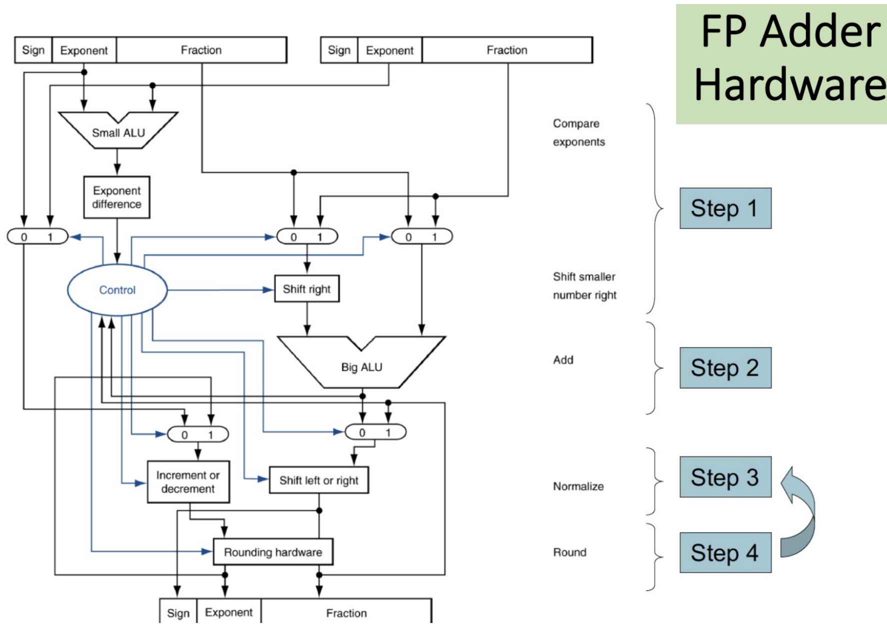
Optimization: can do addition/shift and subtraction/shift in parallel

Floating Point: representation for non integer numbers, allowing small and large representations

- normalized in binary scientific notation
- $(-1)^S * (1 + \text{Fraction}) * 2^{E - B}$
- 32 bits: (31) is sign, (30 - 23) is exponent, and (22 - 0) for fraction; 1 + Fraction = Mantissa
- Overflow occurs when exponent too large and underflow when too small
- Double precision uses 64 bits with 11 bits for exponent and 52 bits for fraction
- Fraction written out as b_1b_2... with implicit 1 (b_1 = 0.5)
- E is normal variable and B is constant (127 for single precision and 1023 for double)
- Exponents 0x00 and 0xFF are reserved for single precision numbers
- Zero represented with S = 0/1, E = 0, F = 0
- Pos/Neg Infinity represented with S = 0/1, E = 0xFF, F = 0
- Not a Number represented with S = 0/1, 0xFF, F != 0
- Denormalized numbers represented with S = 0/1, E = 0, F != 0

Floating Point Addition:

- Align decimal points by shifting smaller exponent
- Add significands
- Normalize result and check for over/underflow
- Round if necessary



-

Floating Point Unit:

- Specialized hardware in separate co-processor for floating point arithmetic
- Allows for floating point to integer conversions
- Operations take several cycles that should be pipelined

	Single-Precision	*Double-Precision*
Addition	`add.s`	`add.d`
Subtraction	`sub.s`	`sub.d`
Multiplication	`mul.s`	`mul.d`
Division	`div.s`	`div.d`
Comparisons Where *xx* can be Example: **c.eq.s**	`c.xx.s` eq, neq, lt, gt,	`c.xx.d` le, ge
Load	`lwc1`	`lwd1`
Store	`swc1`	`swd1`

Also, FP branch, true (**bc1t**) and branch, false (**bc1f**)

MIPS FP Instructions:

FP instructions operate on FP registers (from FPU) at $f0, $f1, ...

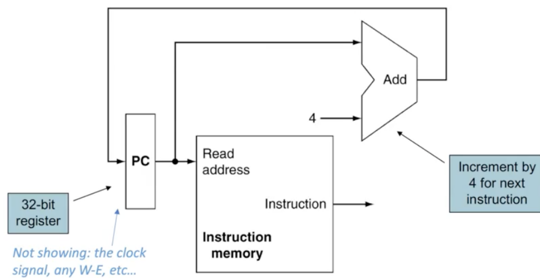
- Floating point registers can be paired for double precision size
- FP addition takes multiple steps; longer than integer addition
- Operations not available for FP: bitshift, modulo

Implementing the Design of a CPU

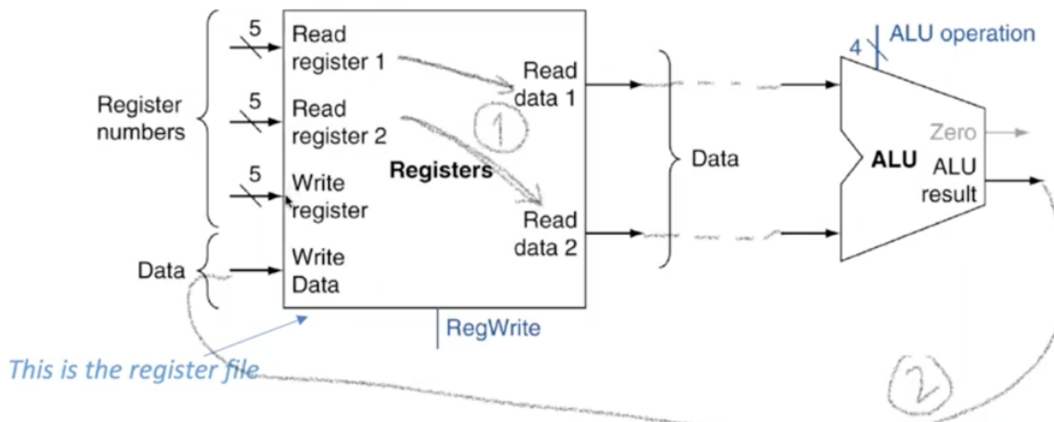
- CPU Performance Factor: Iron Law
    - Instruction Count determined by ISA and Compiler
    - CPU and Cycle Time determined by CPU Hardware
- Consider simple subset for run time
    - Memory reference: lw, sw
    - Arithmetic and logical: add, addi, sub, and, andi, or, slt
    - Control transfer: beq, j

Instruction Fetch-Execute Cycle:

- Send PC to the memory location is and fetch it; then increment PC to next instruction

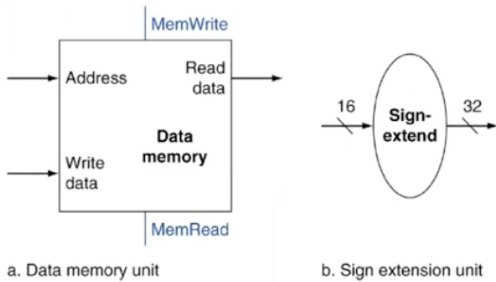


- R Type Instructions:
    - Read the two register operands rs/rt (outputs contents of register)
    - Perform ALU operation (two 32-bit input and 1 32-bit output plus a 1-bit signal if zero)
    - Write result to rd (writes on next clock edge)

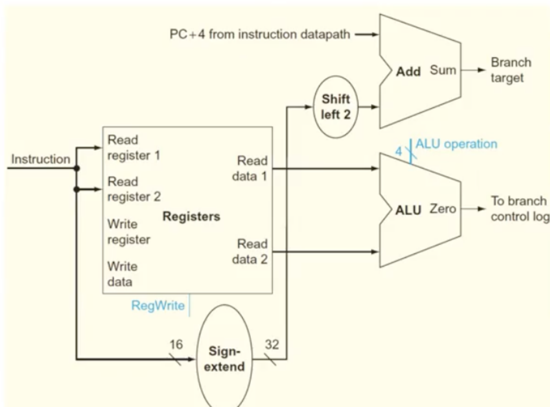


- Memory Reference Classes:
    - Read register operands rs/rt

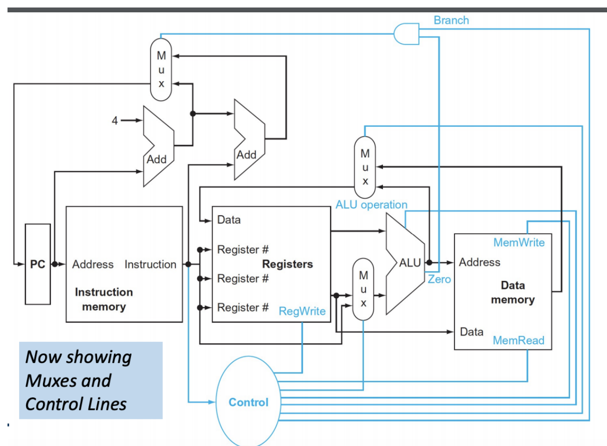
- Use ALU to find offset memory address for ALU
- Load: Read memory and update register
- Store: write register value to memory (on next clock cycle)



- Branching Classes:
  - Read register operands
  - Compare operands (use ALU subtract and check ALU output)
  - Use ALU to calculate target address

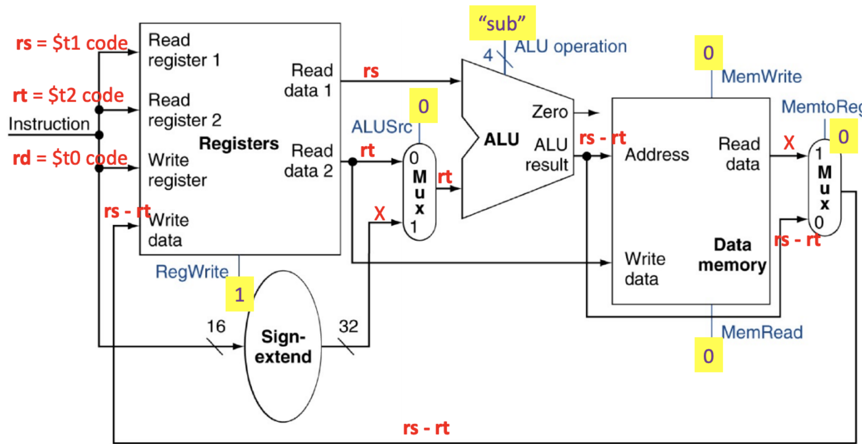


- All these data paths perform one instruction in one clock cycle
- Each datapath element can only do one function at a time
- Use multiplexer when alternate data sources are used



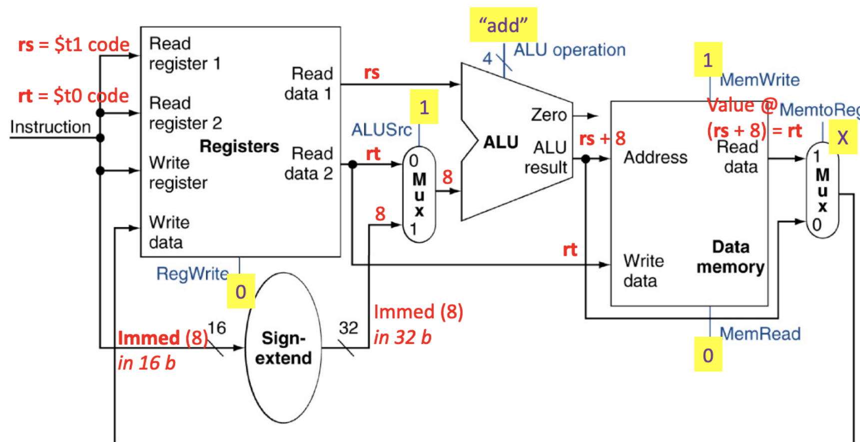
R-Type Datapath

EXAMPLE:
sub $t0, $t1, $t2
R[rd] = R[rs] – R[rt]



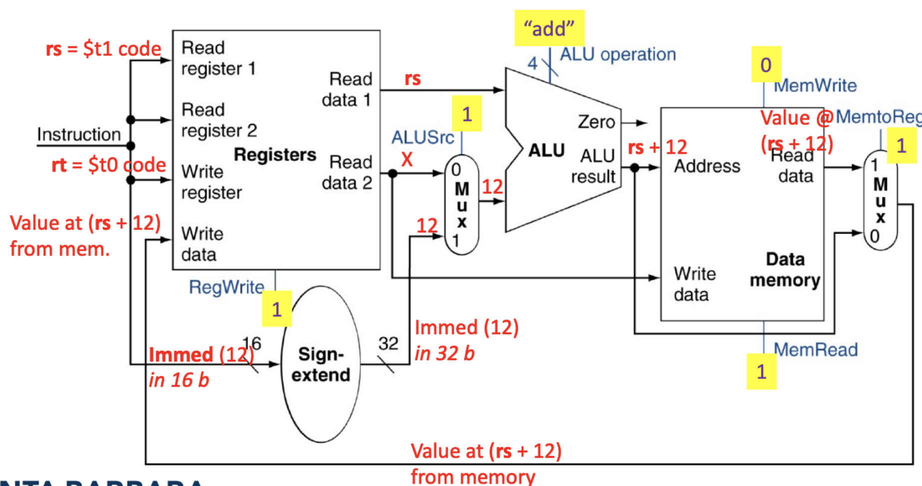
I-Type (Load/Store) Datapath

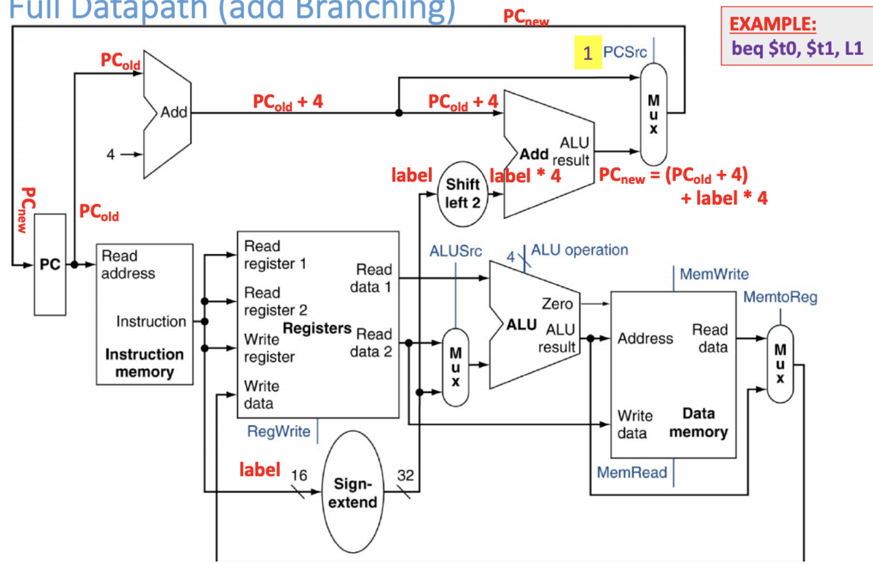
EXAMPLE:
sw $t0, 8($t1)
R[rs]+SignExtImm = R[rt]

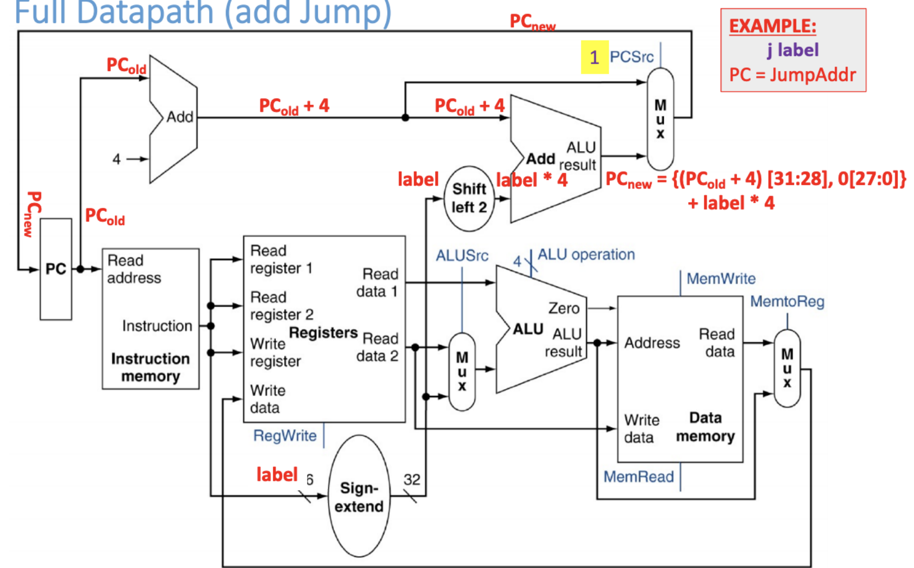


I-Type (Load/Store) Datapath

EXAMPLE:
lw $t0, 12($t1)
R[rt] = R[rs]+SignExtImm

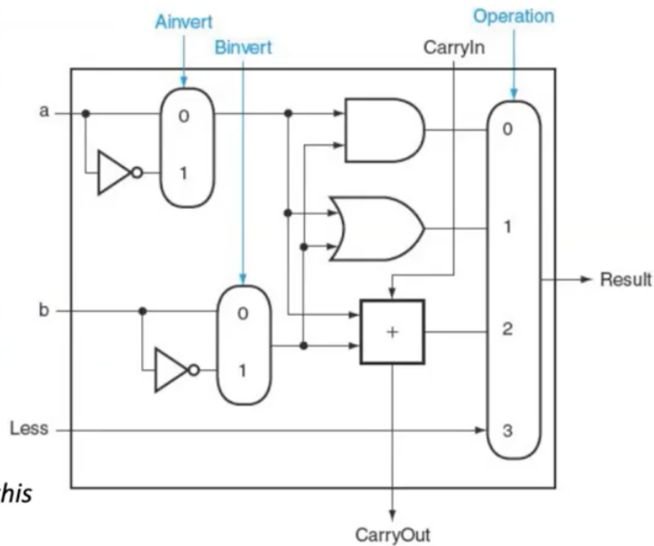



Full Datapath (add Branching)
EXAMPLE:
beq $t0, $t1, L1
PCnew
PCold
1
PCSrc
Add
PCold + 4
PCold + 4
4
M u x
Add
ALU result
label
Shift left 2
label * 4
PCnew = (PCold + 4) + label * 4
PCnew
PCold
PC
Read address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
RegWrite
ALUSrc
4
ALU operation
M u x
Zero
ALU
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
M u x
label
16
Sign-extend
32

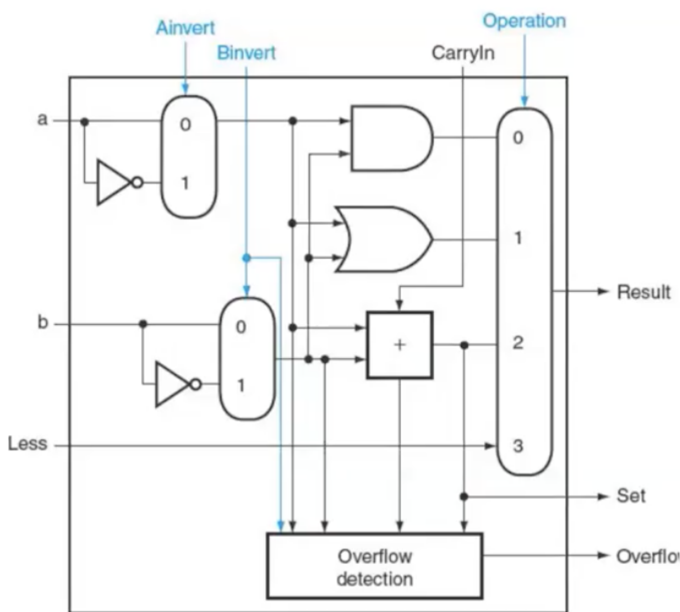

Full Datapath (add Jump)
EXAMPLE:
j label
PC = JumpAddr
PCnew
PCold
1
PCSrc
Add
PCold + 4
PCold + 4
4
M u x
Add
ALU result
label
Shift left 2
label * 4
PCnew = {(PCold + 4) [31:28], 0[27:0]} + label * 4
PCnew
PCold
PC
Read address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
RegWrite
ALUSrc
4
ALU operation
M u x
Zero
ALU
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
M u x
label
6
Sign-extend
32

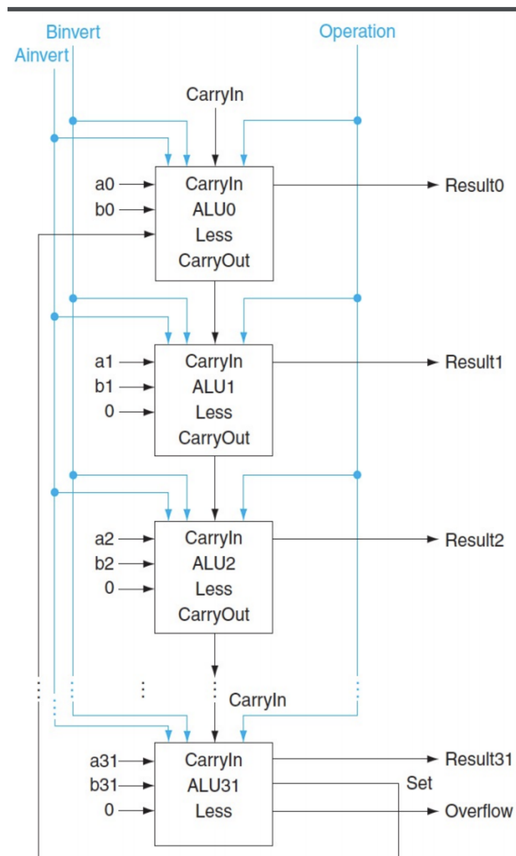
Ainv	Binv	OP[1:0]	Result
0	0	00	AND
1	1	00	NOR
0	0	01	OR
1	1	01	NAND
0	0	10	add
0	1	10	subtract
X	1	11	less*

** **less**: as in a flag for "set-if-less-than". Note that Binv has to be 1 when using this*



Note for subtraction: carry in value must be one





Overflow Detection: check when result of the adder result in overflow (at the end of the cascade)

Another output: set less than, which sets to 1 when true (at the end of the cascade)

Carry in and Binvert work as same way and can be combined into a single bnegate

Subtraction allows equality comparison if zero bit is true

Zero gate only returns 1 if all the result bits are 0 (using a NOR gate)

ALU Control:

- sufficient information from decoded opcode and funct code from instruction to let ALU select
- ALUOp[1:0] (decode of instruction opcode) combined with Instructions[5:0] (the funct field) provides the ALU_Control[3:0] for ALU functionality

a. R-type instruction

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0
				write reg		

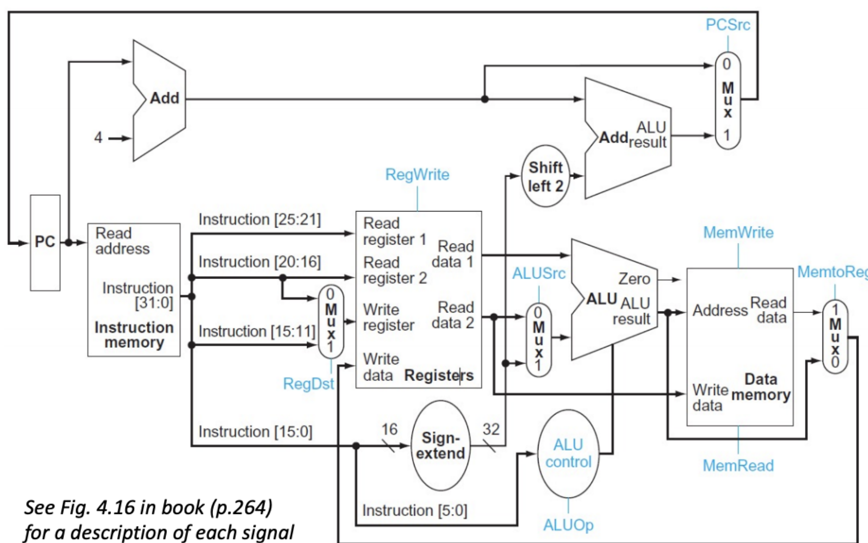
b. Load or store instruction

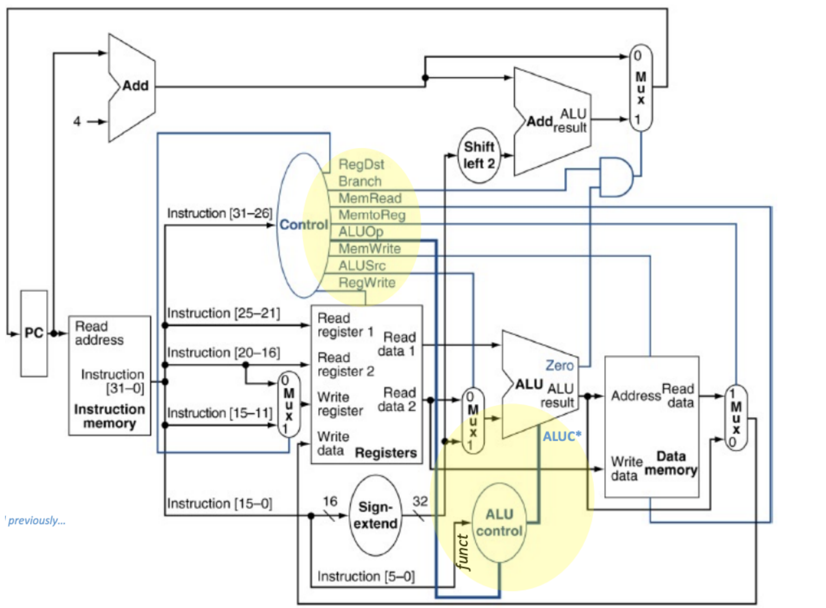
Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0
			write reg	

c. Branch instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

opcode *Op[5:0]* | always read regs | sign/zero extend, sent to ALU and added to stuff...





Performance Issues:

- latencies in the datapath (ignore the wires); sum of latencies is total latency
- iron clad rule: longest delay determines clock period
- critical path: load instruction (longest delay usually)
    - Instruction memory -> register file -> ALU -> data memory -> register file
- not feasible to vary period for different instructions (and complicates design)

Pipelining:

- Pipeline speedup = non-pipeline time / pipeline time = stages
- Stages need to be balanced and allotted to 1 single clock cycle
- Speed is faster due to increased throughput, but instruction latency does not change

MIPS vs Other Pipelining:

- MIPS (and RISC-types) have simpler ISAs versus CISC-types
- MIPS has same length instructions; x86 has variable length instructions
- MIPS has only 3 instruction formats; x86 has more stages bc more format types
- MIPS only has memory access for load store; x86 have other instructions

Pipeline Hazards: prevent starting the next instruction, reducing the performance

- **Structural hazards:** hardware is busy; cannot changed in software (Ex: multiplication for multiple cycles)

- **Data hazards:** new instruction need to wait for previous to read/write
  - Might be able to fix in software (rewrite) or in hardware
- **Control hazard:** deciding on action depends on previous instruction (Ex: branch decisions)
  - Fetching next instruction resolves this issue
  - Resolution: add hardware to compute target early in ID Branch

Forwarding: don't wait for a result to be stored in the register

- Instead, forward the result to a stage that needs in (destination stage later in time than source stage)
- **Pipeline Stall/Bubble:** delay start of next instruction so we can forward to destination stage
- More forwarding = more overhead
- **NO OP:** an stub instruction to fill pipeline that is harmless

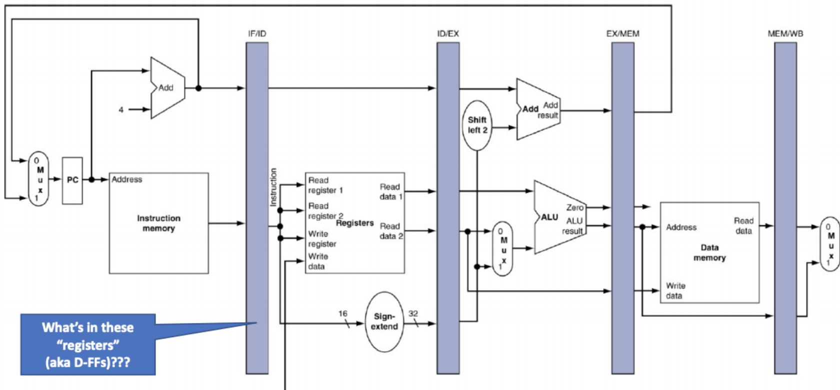
Code Scheduling: reorder code to avoid hazards

Predict the Branch Solution for Control Hazards:

- if prediction is correct, pipeline is not slowed down
- if prediction is wrong, kill the instruction after fetch and is no worse than original bubble
- **Simple Strategy:** predict branches are not taken; only when branches are taken, then we have a bubble
- **Static Predictions:** predict based on typical behavior; take loops but not branches
- **Dynamic Predictions:** hardware measures actual branch behavior based on record of recent history of each branch like a learning algorithm; assume future behavior will continue the trend

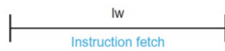
MIPS Pipelined Datapath - Simplified

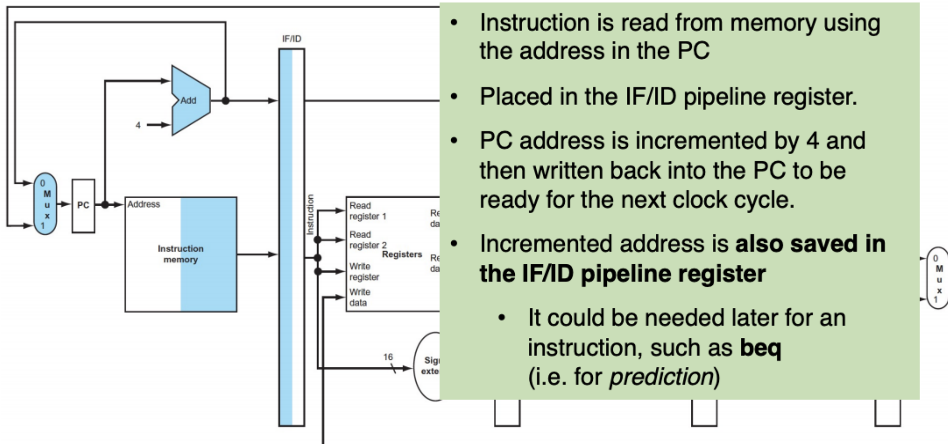
Need registers between stages to hold information produced in previous cycle



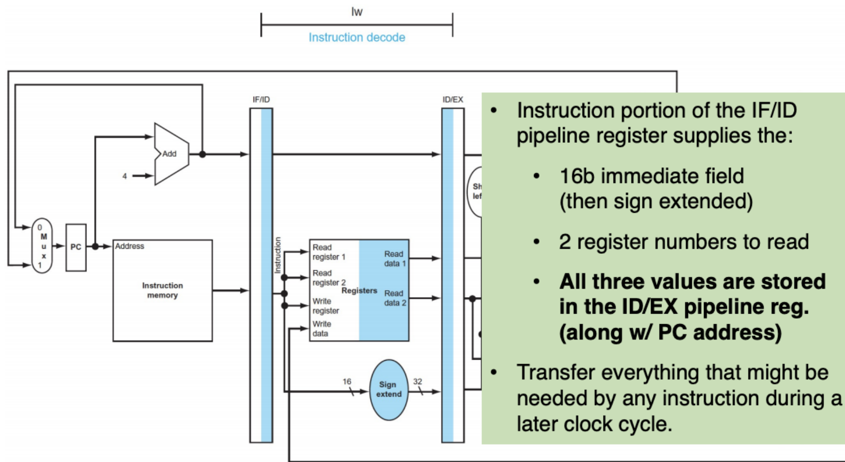
MIPS Pipelined Datapath for lw instruction: IF

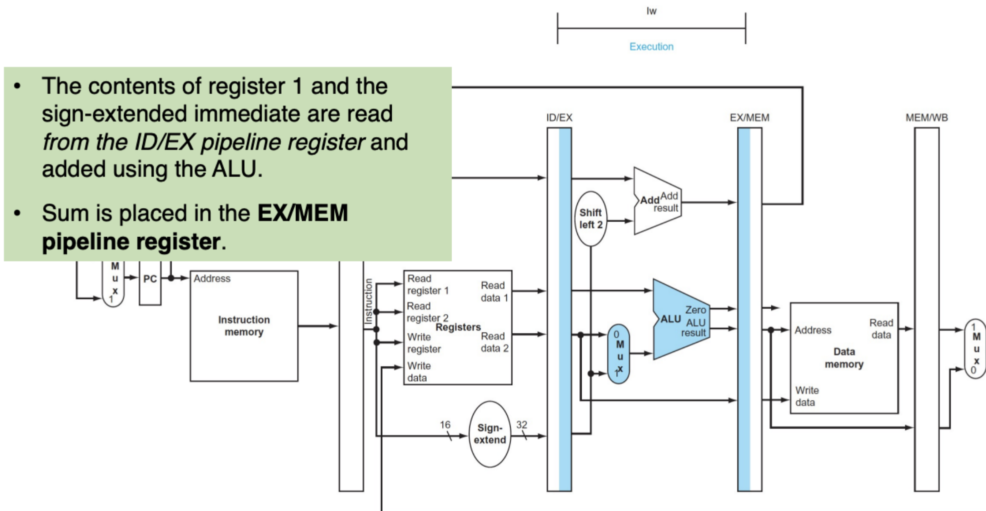
- *Highlight the **right half** of registers or memory when they are being **read***
- *Highlight the **left half** when they are being **written***

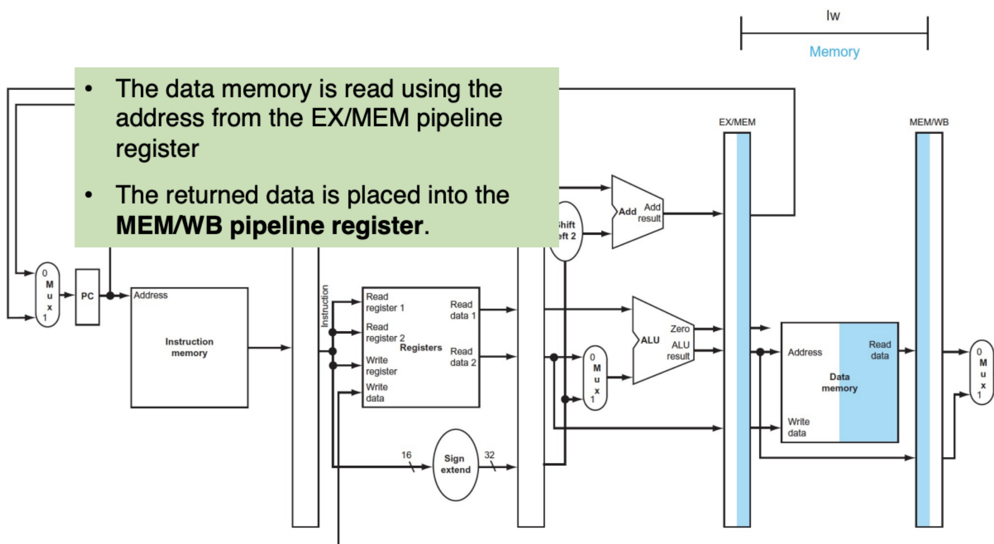


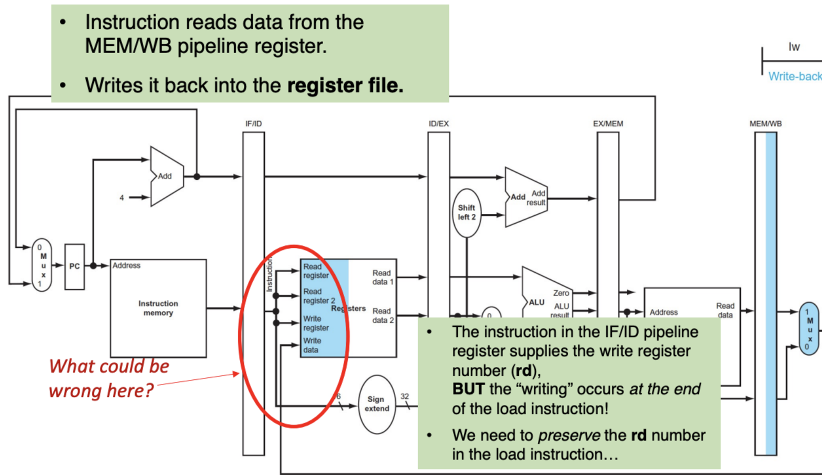


MIPS Pipelined Datapath for lw instruction: ID



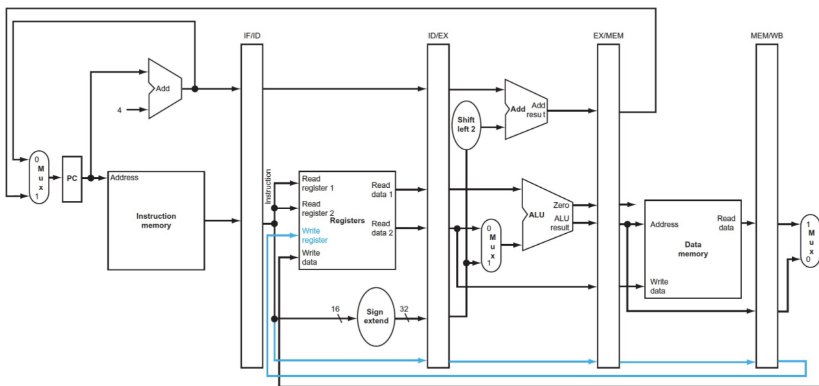






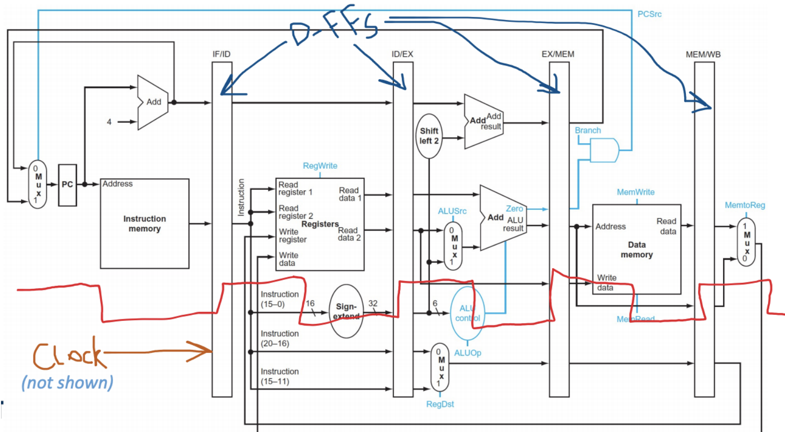
Corrected Pipelined Datapath

Otherwise load instruction wouldn't work properly…

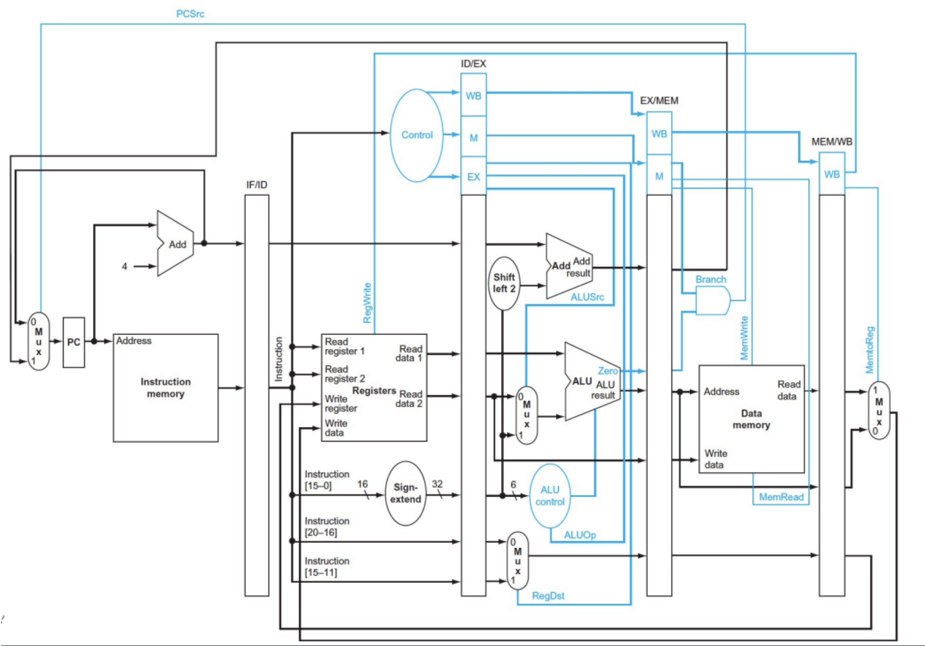


Pipeline Summary:

- One logical component can be only used for a single pipeline stage at a time
- Multiclock Cycles Diagram: good for high level overview and details
- Single Clock Cycle Diagram: full details of 5 instructions



Pipelined Datapath Showing Control Signals



Forwarding vs Stalling: can forward data to any stage that needs it as soon as the write back happens

- Data Hazards Occur When...
  - EX/MEM.RegisterRD = ID/EX.RegisterRS
  - EX/MEM.RegisterRD = ID/EX.RegisterRT
  - MEM/WB.RegisterRD = ID/EX.RegisterRS
  - MEM/WB.RegisterRD = ID/EX.RegisterRT
- Check RegWrite control signal for when registers will be written to
- If RS/RT are the zero registers, RD cannot be overridden (even tho it won't throw an error)
- Note: must stall when read is performed after a load word command writes to a register

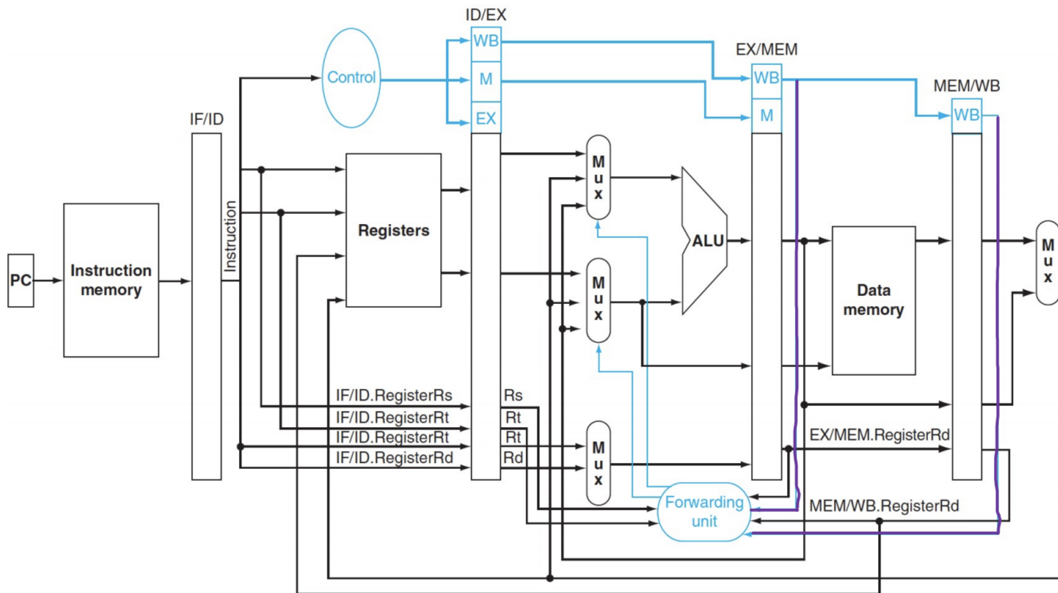
Double Data Hazard: if more than one cause a data hazard, only use the most recent value

EX hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 10
- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10

MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01



Hazard Detection for Stalling:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

How to Stall:

- Force control values in ID/EX to be 0 so that EX, MEM, and WB will do a no-op
- Prevent update of PC and IF/ID register
  - current instruction decoded again, following instruction is fetched again
  - this cycle allows MEM to read data for lw and then can subsequently forward to EX stage
- For compilers to resolve stalls, they must know the hardware info to fix stalling

Branch Types:

- Direct: label has fixed PC; Indirect: label is in register with address
- Unconditional: jump always taken; Conditional: depend on register/immediate values

Control Hazards: want to mitigate latency

- Misprediction gets flushed, rather than an incorrect instruction execution
- **Misprediction Penalty**: amount of cycles needed before branch outcome is determined
- Fixing with compiler with branch delay does not work well due to inaccuracy
- Fixing with compiler using branch hint is also inaccurate

Predictions:

- Branch outcome, determined after execute stage
- Branch address, known after decode stage (direct branches) or execute stage (indirect branches)
- Scheme 1: branch not taken by default; if branch taken, flush away (assume flushing is negligible)
- Scheme 1.5: move branch execution earlier in pipeline
  - move branch adder from EX stage to the ID stage (or add pre-decode stage)
  - evaluate branch decision earlier (harder to do while maintaining short cycle time); need two register reads for equality check using simple gates, but adds more potential hazards
- Scheme 2: based on typical branch behavior (increases to about 70% accuracy)

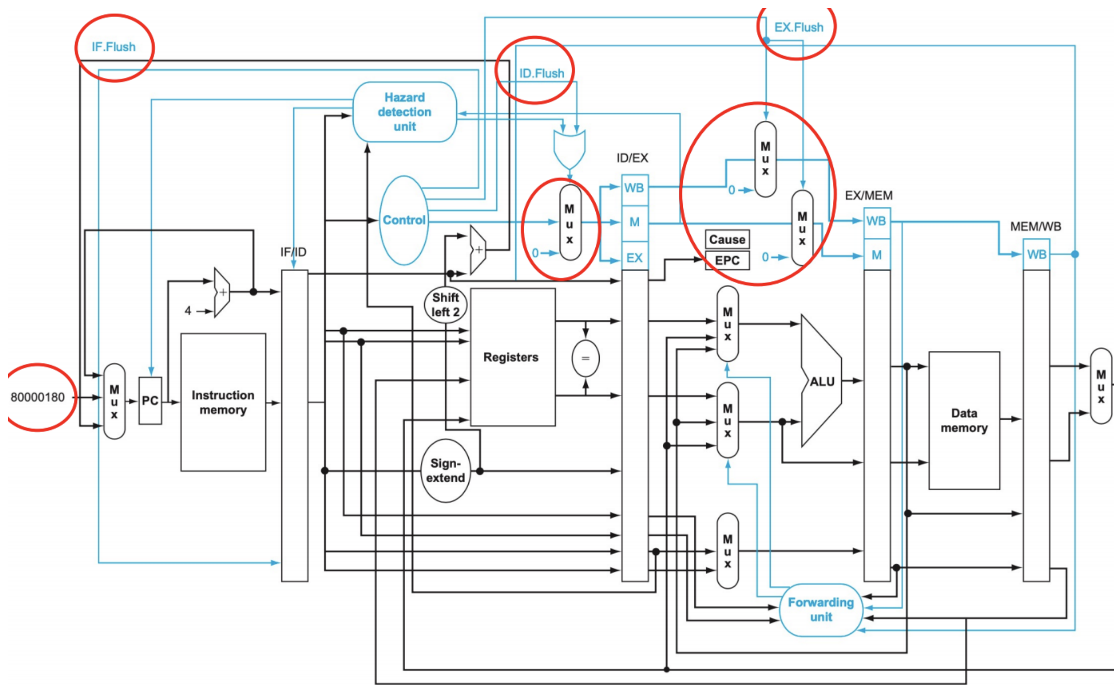
- Prediction schemes not related to the ISA; part of the microarchitecture implementation
- Dynamic Scheme: create branch history table to store past outcome; given a PC, return past result and assume same outcome happens like last time; update prediction if wrong
    - Branch prediction buffer small memory: 1-bit that states take or not take
    - Issue: nested loop construct that can result in a sequence of wrong predictions
    - Two Bit Predictor: only change prediction after two successive mispredictions

Exceptions and Interrupts: unexpected events that require change of flow

- Exceptions arise in the CPU (undefined instructions, invalid addresses, overflow, sycall, break instruction, accessing missing coprocessor, floating point exceptions, etc.)
- Interrupts are from external I/O controller (timers, other processors, external devices)
- Dealing with them without sacrificing performance is hard

Handling Exceptions/Interrupts: need to store issues to get back to original location after handling

- Managed by System Control Coprocessor (CP0)
- Set PC value of instruction in the Exception Program Counter (EPC)
- One method: store problem in a status controller
- Other method: vectored interrupt to go to address based on cause of exception
- **Actions:** transfer to relevant handler; if restartable, take corrective action; otherwise, terminate program
- MIPS treats this similar to a control hazard



Computer Memory: programs access only a small portion of their address space at one time

- **Temporal locality:** if an item is referenced, it will tend to be referenced soon again (i.e., loops)
- **Spatial locality:** if an item is referenced, neighbors will tend to be accessed soon (i.e., array access)
- To take advantage of locality, implement a memory hierarchy
    - everything stored in secondary storage (HDD/SSD) worst case
    - copy recent (and nearby) items to smaller DRAM main memory
    - copy more recently accessed (and nearly) items to SRAM cache memory attached to CPU
    - cache is levelled with higher level at the top
    - block/line = unit of copying (can be multiple words)
    - **Hit ratio:** accessed data in upper level / accesses (hits + absences)
    - **Miss penalty:** time taken to access absent data
    - **Miss Ratio** = 1 - hit ratio

Memory Technology

- **Static RAM** (SRAM): most cache rem is SRAM; 0.5ns - 2.5ns; $1000 - $5000 per GB
- **Dynamic RAM** (DRAM): not in CPU but on motherboard; 10ns - 20ns; $3 - $20 per GB
- **External Storage** (HDD/SSD): secondary; 5 - 10ms; $0.05 GB for HHD; 5 - 100 micros $0.08 per GB SSD

Cache Memory: levels of memory hierarchy closest to CPU

- **Direct Cache Mapping:** each block only has one possible map in the cache
- Scheme: index % blocks (uses lower order bits for indexing)
- Store tags in cache (higher order bits) to reduce storage space
- Add valid bit to confirm whether data is valid (1 = present)
- Temporal locality: new items replace old items if they exist
- Map size must be a power of 2; cache index size is 2^n blocks; n bits used for caching index
- Block (Data) size is 2^m words (2^{m+2} bytes) where m is bits allocated for block offset
- Tag Field is 32 - (n + (m + 2))
- Total number of bits = $2^n * (2^m * 32 + (32 - n - m - 2) + 1) = 2^n * (2^{m+5} + 31 - n - m)$
- Size of a cache excludes the tag / valid field: 2^{m+5}

Block Size Considerations

- Larger blocks reduce miss rate (practical limit to block / cache size)
- For fixed size cache, larger blocks = fewer blocks = increased miss rate

Cache Misses: stall the CPU pipeline; fetch block from next level down; complete data access/restart IS fetch

- have both instruction and data caches
- on cache hit, CPU proceeds normally

Cache Performance: reduces idealness of CPU time

- CPU Time = (CPU execution cycles + Memory stall cycles) * Clock cycle
- Memory Stall Cycles = Miss Penalty * Miss Rate * Memory Access Per Program
- Memory Stall Cycles = Miss Penalty * Misses Per Instruction * Instructions Per Program
- **Average Memory Access Time** (AMAT) = Hit Time + Miss Rate * Miss Penalty
- When CPU performance increases, miss penalty become more significant since hardware needs to improve for fetching; decreasing base CPI/increasing clock rate does not fix memory stalls
- **Compulsory Miss:** first use of a block
- **Capacity Miss:** cache is too small
- **Conflict Miss:** collisions due to less than full associativity

Multi Level Caches:

- Level 1 primary cache attached to CPU (small, fast)
- Level 2 larger slower but faster than DRAM
- L1 focuses on minimizing hit time for shorter clock cycle; L2 cache focuses on reducing miss penalty

Cache Associativity:

- Memory access patterns can cause CPU to repeatedly access addresses to map to the same index
- One location = direct map; Any location = fully associative; N location = set associative
- Fewer index bits used for same cache size
- Data in each cache set is differentiated by tags (tags looked at in parallel)

Virtual Machine: emulation method of a standard software interface

- A host computer emulates a guest OS and machine resources
- Apps can't tell they are running in a VM
- Improves isolation of multiple guests, sharing (cloud computing); users are isolated from each other
- A single computer can run several different OSes
- Additional layer of security beyond standard OS
- Computers are fast so virtualization isn't too expensive

Virtual Machine Monitor (VMM)/Hypervisor: software that supports VMs

- Maps virtual resources to physical resources (memory, i/o devices, cpus)
- Guest code can run on native machine in user mode
- Allows guest OS to be different from host OS

Hypervisor Requirements:

- Guest software should behave on a VM exactly as on native hardware
- Guest software should not be able to change allocation of the real system's resources directly
- Hypervisor must be at a higher privilege than the guest (subset of instructions for system)

Options for Hypervisor: virtualization has overhead

- When guests access a device, the hypervisor must step in
- Provide illusion of real device is tricky (need to cause exception with every access)
- Can provide fake devices that are easier to map to hypervisor
- Can adopt para-virtualization (API for guests, additional to existing ISA)
- **Note:** recursive virtualization is possible with ISA/hardware support

Virtual Memory: not virtual machines

- **Translation:** virtual addresses into physical addresses by CPU and OS
- **Protection:** application permissions to access data at an address
- **Disk Cache:** use main memory as cache of secondary storage
- Programs share main memory
- Each application gets private virtual address; protected from other programs
- Takes control of memory addressing and lets each application function as it had unlimited memory

Memory Pages:

- Translations stored in page tables, with key = virtual address and value = mapping to physical address
- Takes 4 bytes per data to store, so storing every address would take $4 * 2^{32}$ space
- If we only store words, it still takes the same amount of space as physical memory
- Want large page size large so the page table can be smaller

Virtual Addressing: Virtual Page Number + Offset gets translated to Physical Page Number + Offset

- Upper portion of address is page number, lower portion is page offset
- Page Size = $2^{\#\text{ of page offset bits}}$
- Does not need 1:1 mapping with physical addresses
- Loading virtual addressed with valid = 0 causes page fault

Page Faults:

- page must be fetched from secondary storage (HDD/SSD)
- can take millions of clock cycles; handled by OS code

- want to minimize page fault rates by making pages large enough, optimize organization of memory, involve smart algorithms for retrieval
- page table entries have tags and valid bits too; when valid bit is off, page fault
- usually OS keeps swap space in secondary storage with pages from memory
- OS manages which processes/addresses use each physical space
- If replacements need to be done, some prediction scheme has to be used (like branching)

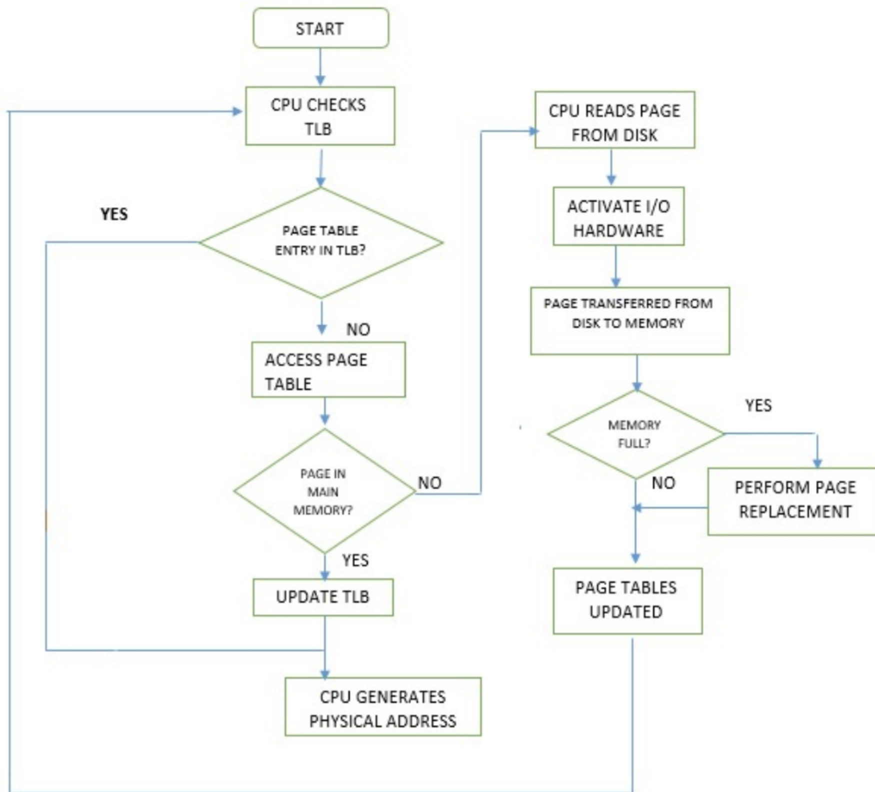
Write Through vs Write Back

- When writing to memory and that memory location is a copy of something further down:
    - **Write Through** if write to both cached copy and original place in memory
    - **Write Back** if writing to only cached memory and doing rest later
- **Dirty Bit:** indicate if data present in the cache/page was modified (Dirty); for write back only
- To reduce page fault rate, prefer least recently used (LRU) replacement
    - Reference/use bit in page table set to 1 on page access
    - Periodically cleared to 0 (not used recently) by OS
- Disk writes take million of cycles, write block at once not individual locations

Translation-Lookaside Buffer (TLB): cache of valid page table entries

- CPU include a special cache of recently used translations (different from instruction/data cache)
- Usually fully associative with 16-128 entries
- TLB reach = # of entries * page size (how much memory can be read without missing)
- TLB meant to make virtual memory access faster
- If TLB too small or too many processes in use, thrashing occurs: frequent TLB misses, new cached page displaces one that will be soon used again, degrade overall performance
- Used before instruction fetch and data load/store since those contain virtual addresses

Hierarchical Page Tables: make structure smaller by making levels to split it up, but causes multiple accesses


START
CPU CHECKS TLB
PAGE TABLE ENTRY IN TLB?
YES
NO
ACCESS PAGE TABLE
PAGE IN MAIN MEMORY?
NO
YES
UPDATE TLB
CPU GENERATES PHYSICAL ADDRESS
CPU READS PAGE FROM DISK
ACTIVATE I/O HARDWARE
PAGE TRANSFERRED FROM DISK TO MEMORY
MEMORY FULL?
YES
NO
PERFORM PAGE REPLACEMENT
PAGE TABLES UPDATED

-