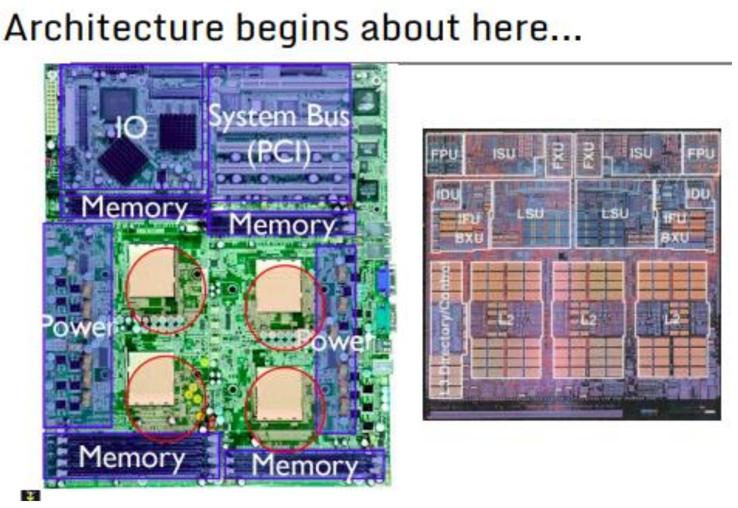
Architecture

CSC2002S

Introduction



- Hardware design has a huge impact on software
  - Memory hierarchy - relevant data should be nearby
  - Multi-core- thread management, how to write smarter multi-threaded programs?
  - Instruction formats

Trends

PROCESSOR TECHNOLOGY TRENDS

- Processor's made up of transistors
  - Formed as integrated circuit by combing many transistors together
- Shrinking of transistor sizes: 250nm (1997) ➔ 130nm (2002) ➔ 70nm (2008) ➔ 35nm (2014)
- Transistor density (on a chip) increases by 35% per year and dye size increases by 10-20% per year… functionality improvements!
  - Can put more transistors on a smaller space.
- Transistor speed improves linearly with size – smaller transistor has greater speed

MEMORY AND I/O TECHNOLOGY TRENDS

- DRAM density increases by 40-60% per year, latency has reduced by 33% in 10 years, bandwidth improves twice as fast as latency decreases
  - If have data on DynamicRAM (DRAM) memory, means that when you need to read from there, can go to any point on that circuit and read the data on that part.
  - This contrasts to moving storage (e.g. hard disks) – to read data, need to move needle that reads data from disk.
- Disk density improves by 100% every year, latency improvement similar to DRAM

Eight Great Ideas in Computer Architecture

- Design for Moore's Law
  - Computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts.

- Use Abstraction to Simplify Design
    - A major productivity technique for hardware and s-ware is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels.
- Make the Common Case Fast
    - Making the **common case fast** will tend to enhance performance better than optimizing the rare case.
- Performance via Parallelism
    - Perform opns in ||
- Performance via Pipelining
    - Particular pattern of parallelism - **pipelining**.
    - Example: forming a human chain and passing a bucket of water along it to put a fire out (more efficient than having individuals run back and forth).
- Performance via Prediction
    - In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate.
- Hierarchy of Memories
    - Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost.
    - Architects have found that they can address these confl icting demands with a **hierarchy of memories**, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom.
- Dependability via Redundancy
    - Computers not only need to be fast; they need to be dependable.
    - Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures.

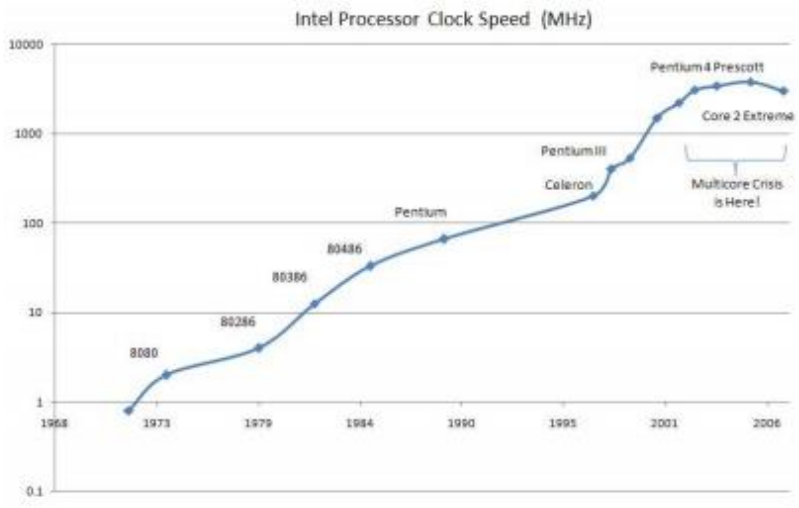
Power Wall

- Major block to increases in processor performance
- Electric energy consumption of chip is limiting factor for increasing processor speed and frequency
- Overall processor performance is generally increased if transistors in integrated circuit can be switched on and off faster – since this means more processing is being done.
    - Remember that transistors consume energy when they switch states – energy consumed during this process is ***dynamic energy***.
        - Dynamic energy is directly proportional to capacitive loading and voltage applied to the transistor.
            - DE = proportional to capacitive load in each transistor, multiplied by voltage squared.
        - Capacitive load: energy that's persistently stored in a device, such as a transistor
            - Enables transistor to maintain its operation.
            - In the case of transistors, has to maintain a lag between current and voltage (when voltage and current goes up - create lag there)

- Dynamic power consumption proportional to dynamic energy x transistor switching frequency
- Dynamic Power: α
  - activity x capacitance x voltage x frequency
  - these factors crucial in determining how much power processor/chip is using.
- Power/energy that comes through to chip/processor is dissipated as heat and thus must be removed.
  - If heat accumulates faster than it can be removed from chip, processor will burn.
    - Cooling technology is very expensive.
  - Thus, increase in chip temperature slows down transistor switching rate and thus slows down processor speed.
  - Have to thus ensure chip does not overheat – at the cost of performance.
- Power wall: limitation placed on processor clock/frequency and overall CPU performance improvements.
  - Limit is because have to be able to deal with heat constraints that comes with power processing and each processor.
- Voltage and frequency are somewhat constant now, while capacitance per transistor is decreasing and number of transistors (activity) is increasing.

Multi-core Microprocessors

- Power limit forced dramatic change in design of microprocessor
- Instead of trying to make a single processor faster, new paradigm was moving towards microprocessors and multiple processors on a chip – multicore microprocessors.



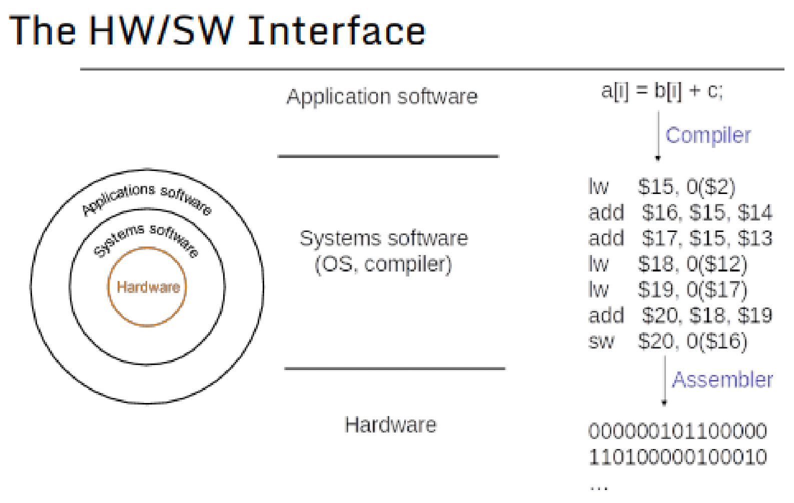
- Had to refactor code to take advantage of increasing processors

WHAT DOES THIS MEAN TO A PROGRAMMER?

- Programmers have to be aware of parallel hardware and need to write programs to be || in order to get improved performance
- Improvement requires the program to be multi-threaded
- Threads need efficient synchronization and communication

- Data placement in the memory hierarchy is important – time taken to get data to processor should not become bottleneck in performance.
    - Want to put data as close to processor as possible; and move data before it's required by processor.

Hardware-Software Interface



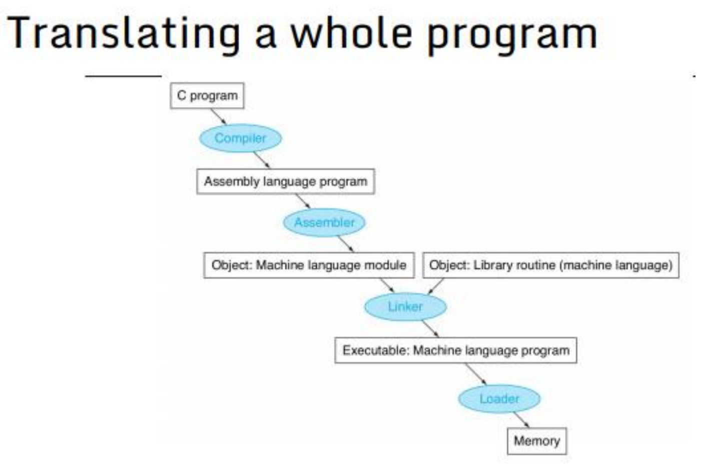
```
a[i] = b[i] + c;

lw   $15, 0($2)
add  $16, $15, $14
add  $17, $15, $13
lw   $18, 0($12)
lw   $19, 0($17)
add  $20, $18, $19
sw   $20, 0($16)

000000101100000
110100000100010
...
```

- Note: hardware in comp can only execute low-lvl instructions
    - Means need layers of s-ware to interpret/translate high-lvl programme code into simple computer instructions
- Computer h-ware works directly with systems software.
    - OS and compiler are two systems software central to opn of computer
    - OS has "supervisor" role – handles I/O, allocating memory, managing simultaneous access to shared resources, and so forth
    - Compilers – translate high lvl prog into low lvl instruction that h-ware can execute
        - This low lvl is in the form of bits
- Processor is just a circuit with transistors (that just switch on/off) – translates well to bits (0 or 1).
    - Binary language is machine language that h-ware can understand and execute based on its circuit.
- Assembler translates from assembly to machine language.

MOVING DOWN THE LAYERS

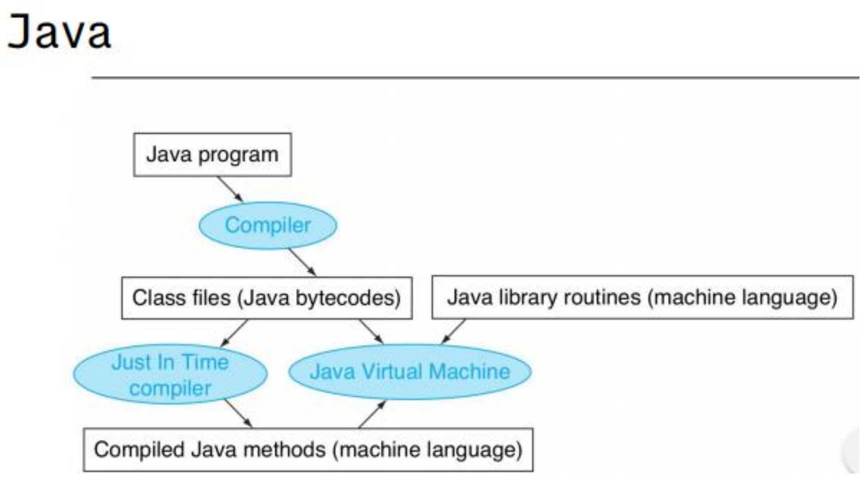
- Translation from high lvl to binary – shown in two steps, but can be done in one
    - Compilers cut out middle-man – translate to binary directly
- Compilers translate machine specific machine code
    - Sometimes OS dependent, because different OSes have different ways of dealing with instructions.
        - Compiler would thus create machine language that would only run on specific platform that compilation has been done for
            - Key fact about compilers

COMPILING PROCESS


Translating a whole program
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

- E.g. Have high lvl program running through compiler into assembly language
    - Then have assembler which takes assembly language into machine language which may be specific to the machine
    - Then, to execute program, need to use *linker*, which connects with routines and libraries available on that machine to make that program complete i.e. able to run.
    - Linker is **almost always** OS specific – but makes it possible for compiled program to run on different OSes.
        - E.g. could have program that's not compiled but has been linked, run it through linker and it'll result in a machine language program that's very specific for h-ware on which you're running.
- Not all compiler systems separate compiler from linker – without separation, then resulting compilation will be very h-ware specific.
    - Can't compile and run it on another machine

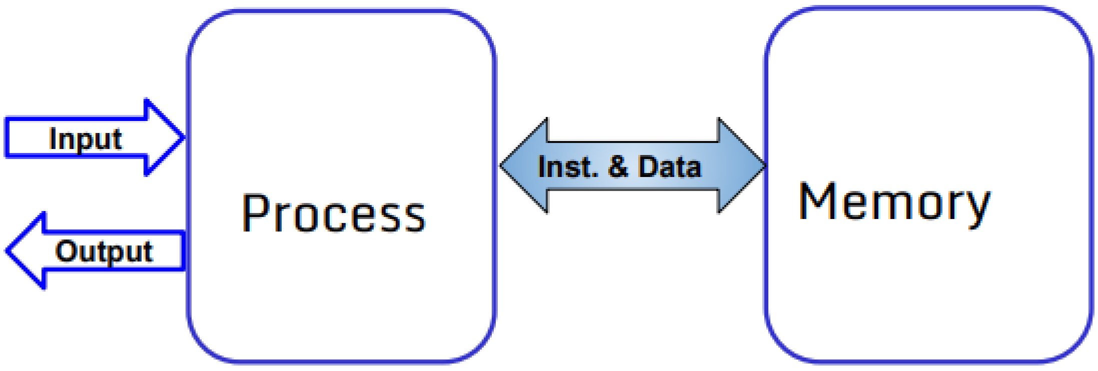
In Java


Java
Java program
Compiler
Class files (Java bytecodes)
Java library routines (machine language)
Just In Time compiler
Java Virtual Machine
Compiled Java methods (machine language)

- Java compilation components – ARE OS and machine independent
- Generate **byte code** – code then run on JVM
- Once compiled, would be able to run on almost anything that can run JVM.

- JVM would need Java library routines – these are machine dependent
    - Once run through Java machine, then becomes dependent, and so can't run on another machine.
- Byte code can be run on any machine with JWM, but *output* of JVM would be very machine specific.
- Alternative – Just In Time compiler
    - This compiler runs after program has started and runs byte code on the fly
        - Doesn't necessarily run through JVM – but runs direct on machine, compiles it
            - Gets faster output, typically running hosts own native instruction set
                - Very specific to computer on which it runs.

Assembly Programming



- What makes a computer? Input -> Process -> Output (simply)
    - Maybe externally, but process differs internally
- Internally- process involves other steps
    - E.g. data we give to process, might have to be stored internally in some form of memory; as processing is happening, will be fetching data from memory, sending it into process, writing result into memory; then fetch more data and cycle continues.
- Note: data/input going to process is going to include both instructions and data
    - In most cases, data will be stored in memory and processor will be interacting with memory for the duration of the processing until output sent back to external part of system

VON NEUMANN

- Key insight: instructions are data! (as far as processor is concerned)
    - E.g. a compiler receives a program and works on that program to receive some output.
- Program can manipulate other programs
    - Compilers, debuggers, Operating systems
    - Reflection classes in most languages (incl Java)

- One major ‘issue’ – von Neumann bottleneck (in processing):
    - A processor is idle for a certain amount of time while memory is accessed – it’s waiting for data to be fetched from memory
        - E.g. receiving an instruction which needs to be fetched from memory
    - Presents a significant problem – if processor very fast, but time takes to get something out of memory into processor is very long, then that’s going to affect overall processor performance.

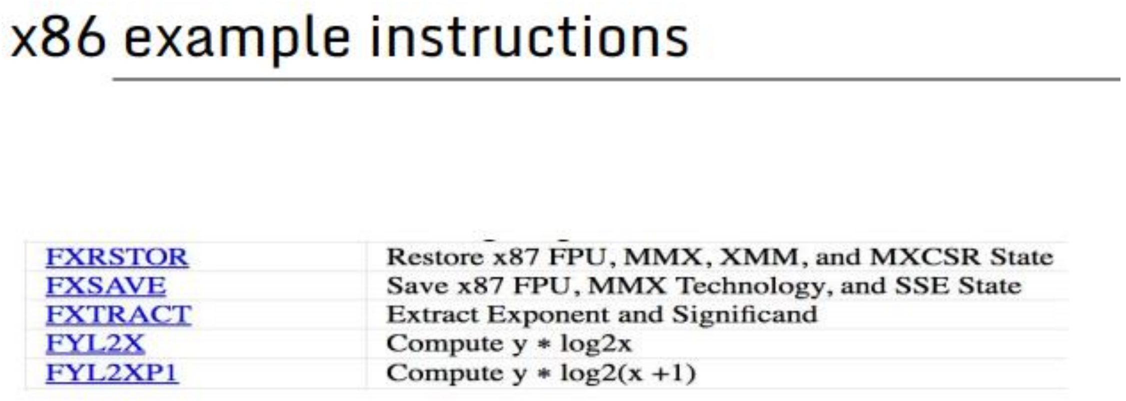
Standard Operation

- Fetch-execute cycle:
    - **Fetch** next instruction from memory
        - Increment program counter – pointer to memory indicating where next instruction is
            - Usually increment sequentially to get next instruction in memory
            - Sometimes prog counter may have to jump to next instruction in memory (non- sequentially)
    - **Decode** instruction
            - Each processor has specific instruction set it operates on
        - Read operands
            - These are variable instruction needs to execute
            - E.g. if opn is an “add”, then get values we’re going to add together and the variable to store the result.
    - **Execute** operation
        - Write results to memory

INSTRUCTIONS

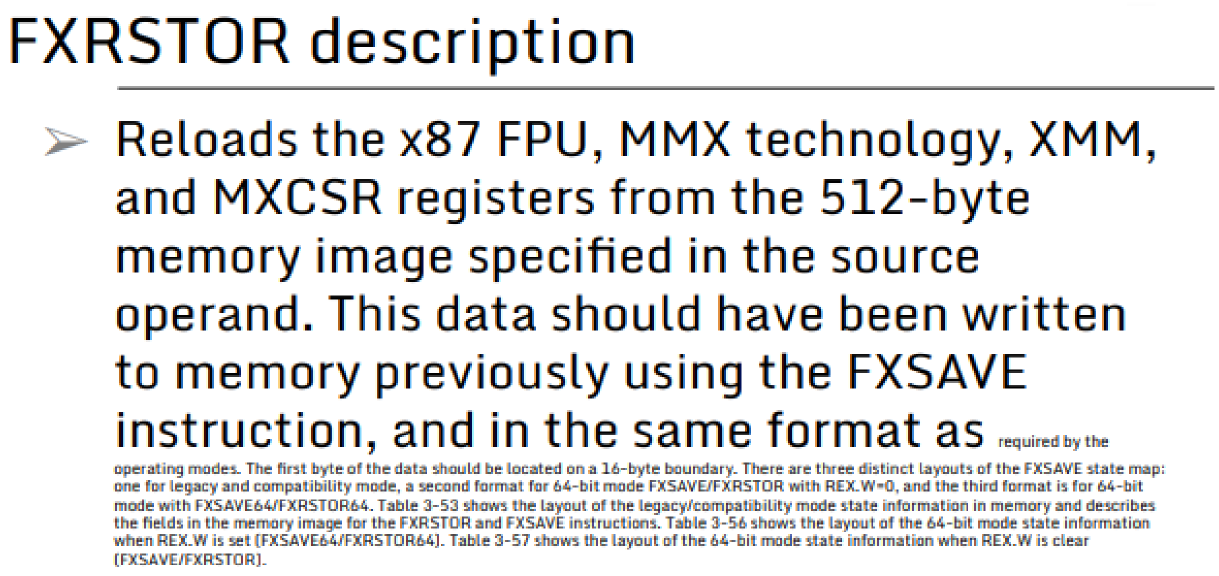
- Two basic philosophies
    - Complex Instruction Set Computing (CISC)
        - Older architectures – VAX, x86
        - Followed format of high-level languages
        - One instruction does a lot
            - Each instruction can be complex and within processor, several tasks may need to be done in order to fulfill instruction

        - CPU designed in a way that several parts *must* interlock when it receives a particular instruction – needs to worked together like a “pipeline” in order to accomplish that instruction.



x86 example instructions

FXRSTOR	Restore x87 FPU, MMX, XMM, and MXCSR State
FXSAVE	Save x87 FPU, MMX Technology, and SSE State
FXTRACT	Extract Exponent and Significand
FYL2X	Compute y ∗ log2x
FYL2XP1	Compute y ∗ log2(x +1)



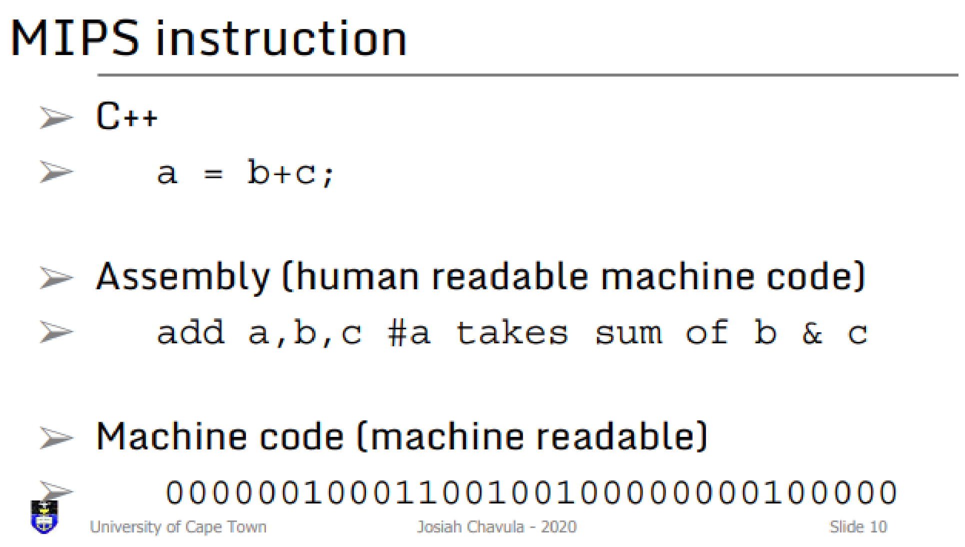
FXRSTOR description

➢ Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-53 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-56 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-57 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

    - Reduced Instruction Set Computing (RISC)
        - Simple instructions executed rapidly
            - Small unit tasks that can be achieved by processor in single step
        - Modern processors – MIPS, PowerPC, ARM

Reduced Instruction Set Computing

- MIPS: Microprocessor without Interlocked Pipelined Stages
- Each instruction must be achieved by single component within CPU – without use of interlocking
- Clean, scalable design because of lack of interlocking

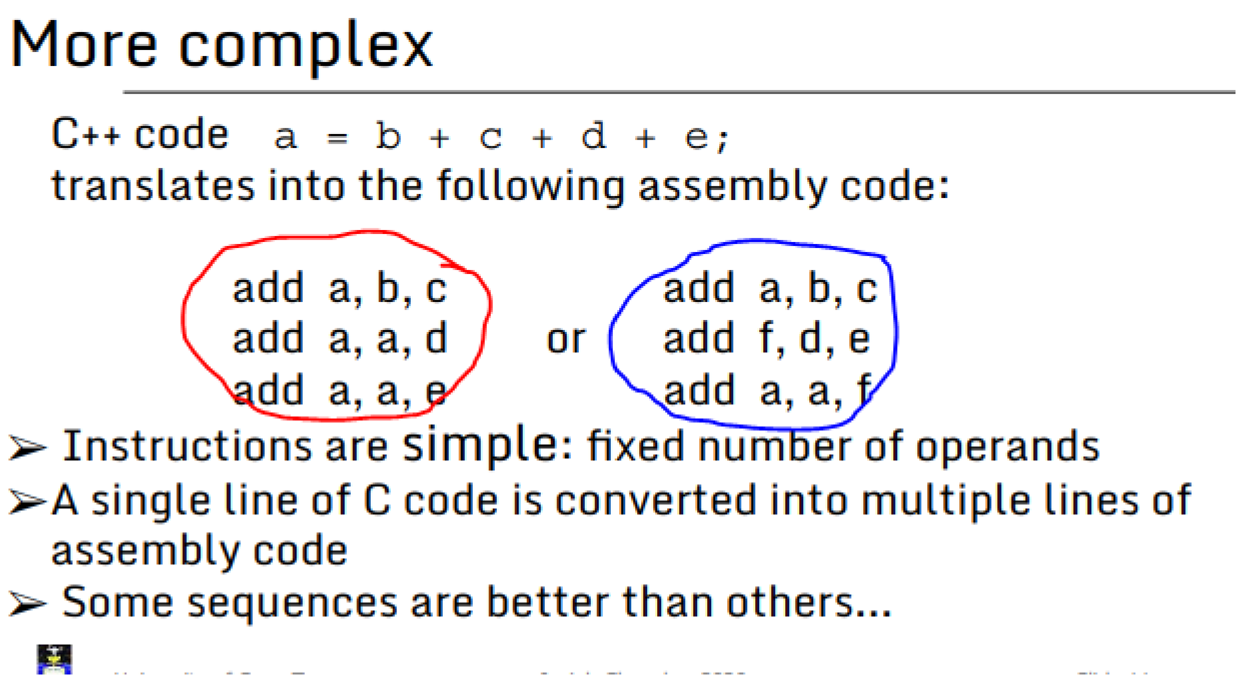


MIPS instruction

- C++
- `a = b+c;`
- Assembly (human readable machine code)
- `add a,b,c #a takes sum of b & c`
- Machine code (machine readable)
- `00000010001100100100000000100000`

University of Cape Town Josiah Chavula - 2020 Slide 10

- Have assembly language statement – take result of b and c and put into a
    - That instruction will be translated by assembler into machine code
- Instructions (add) and operands (a,b,c) will be translated to binary representation in machine code
    - Add: 100000



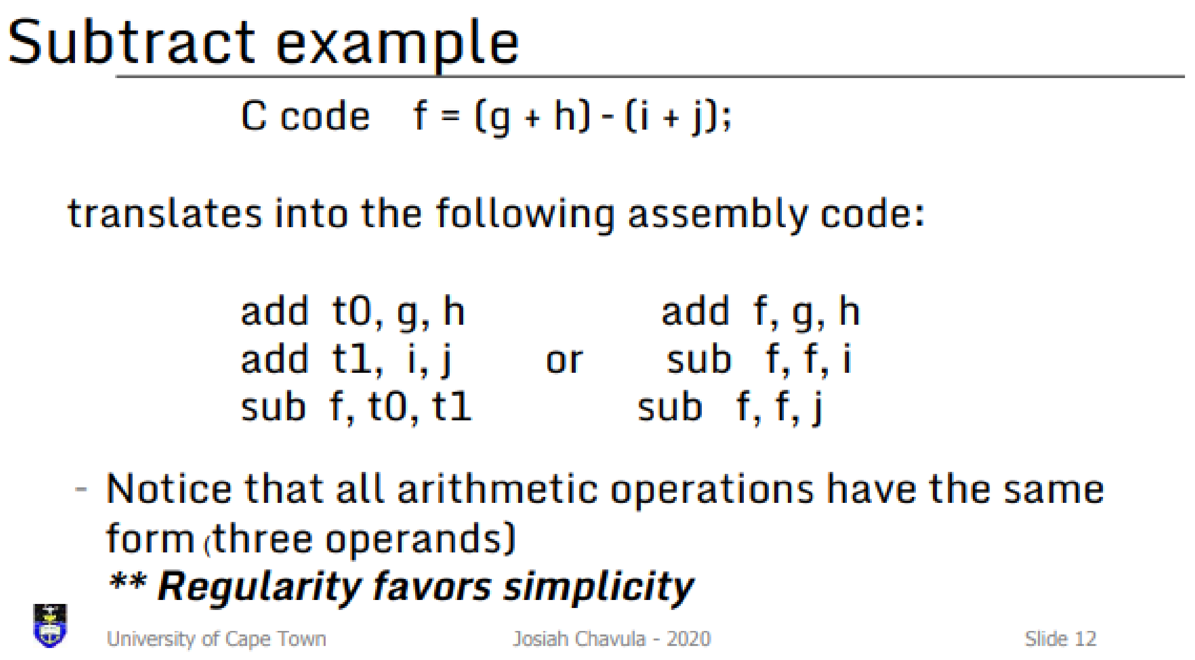
More complex

C++ code `a = b + c + d + e;`
translates into the following assembly code:

```
add a, b, c          add a, b, c
add a, a, d    or    add f, d, e
add a, a, e          add a, a, f
```

- ➢ Instructions are simple: fixed number of operands
- ➢ A single line of C code is converted into multiple lines of assembly code
- ➢ Some sequences are better than others...

- For RISC – can't have whole computation in one instruction
  - For MIPS, can't have more than 3 operands (processor won't be able to understand it).
  - So, have to take instruction and decompose it into multiple stages that can be understood and be computed by CPU as unit tasks.
- A single line of high-level programming language code is converted into *multiple lines* of assembly code
- When looking at solution, have to consider design and performance perspectives
- Blue solution: uses additional variable, f
  - Because of temporary variable, can now do [add a, b, c] and [add f, d, e] *at the same time* because those results are going to be assigned to different parts of memory.
    - Can be easily pipelined
  - Processors implement pipelining – can have certain set of instructions running in || to each other, so long as instructions don't depend on each other
- Red solution: all steps depend on a
  - Second step needs to wait for first to complete in order to execute
  - Use - remember, have limited # of memory locations available for running program; and so, may not always have the option to use extra variables (as with blue solution)



Subtract example

C code f = (g + h) - (i + j);

translates into the following assembly code:

```
add  t0, g, h            add  f, g, h
add  t1,  i, j     or     sub  f, f, i
sub  f, t0, t1          sub  f, f, j
```

- Notice that all arithmetic operations have the same form (three operands)
  ***** Regularity favors simplicity***

University of Cape Town Josiah Chavula - 2020 Slide 12

- An important principle: Regularity favors simplicity
  - If the same instruction always has the same form, it makes it easier to implement in the processor
    - E.g. if processor gets “add” instruction, it “prepares itself” to read a set # of operands.
  - Easier to design processors when instructions are definitive
    - If they weren’t, processor for e.g. would need different mechanisms to deal with adding 3 numbers compared to 5

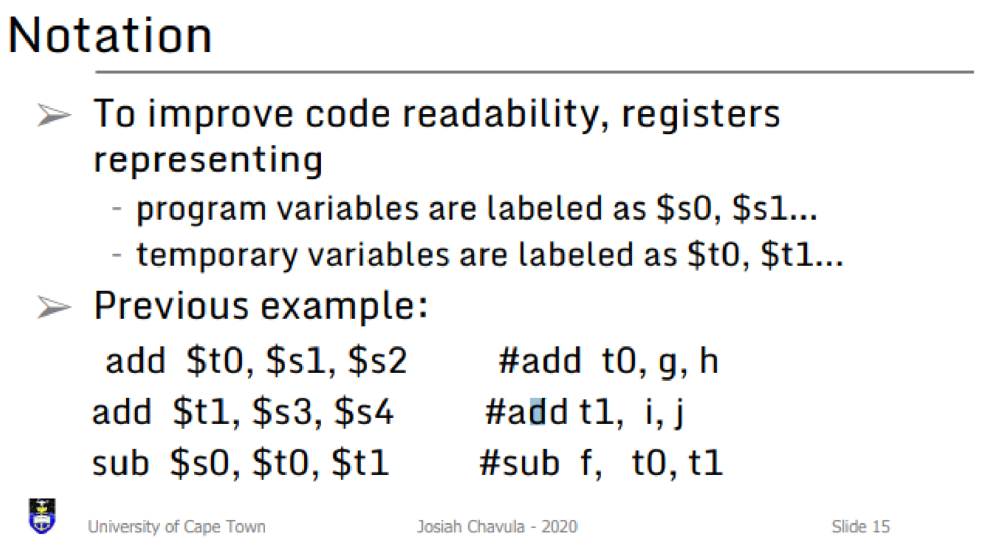
VARIABLES

- A variable is a location in memory
  - But RAM is slow (expensive) to access
- So, processor contains several high-speed memory slots on-chip called *Registers* which are used to hold variable values
  - There are a fixed (small) number of these – but they’re very fast
  - Register instruction set only works on values that are already in the register
    - So before can process anything, it must be moved to registers from main memory
- RISC instructions only work on registers (big departure from CISC)
  - CISC does allow processing of values that are still in main memory

REGISTERS

- MIPS is a 32-bit architecture, meaning all instructions, addresses are 32 bits long (4GB RAM limit)
  - Most modern architectures are 64-bit architectures, which have 64 bit words, support 18.1 exabytes RAM
- MIPS processors (and most architectures) have 32 registers
  - Each register is 32 bits wide
  - 32 bits = 4 bytes = 1 word

- Why 32?
  - For each register, need to be able to identify/represent them in the instruction
  - If have instruction made up of 32 bits, need some of those bits used to represent register that you're pointing to by that instruction
  - If have 5 bits ($2^5 = 32$), means need 5 bits to represent 1 register
  - If had more registers, means we'd need more bits to represent each of the registers
    - If we had 64 registers, need 6 bits to represent each of the registers – if we need 3 registers for an instruction, we loose 18 bits just for pointing to registers
  - It takes away from bits that could be used for instruction – thus need to balance things.



Notation

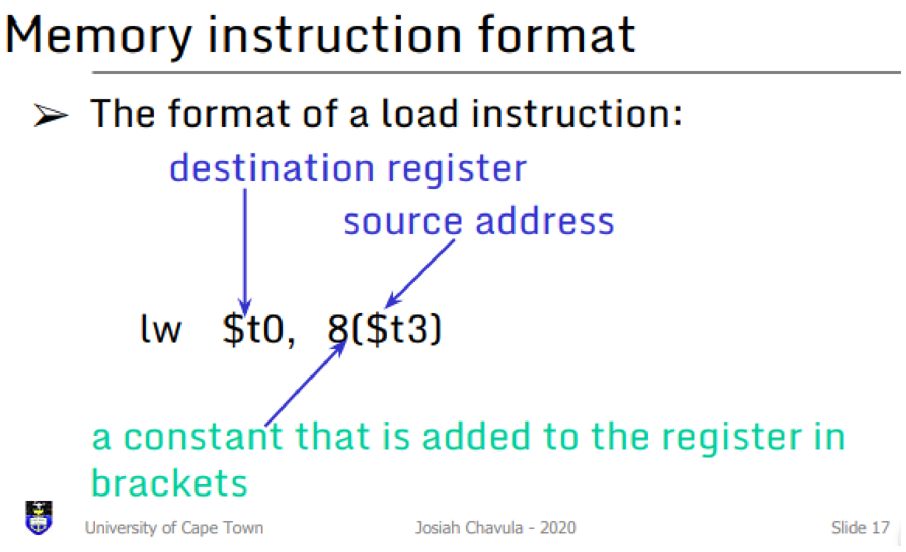
- To improve code readability, registers representing
  - program variables are labeled as \$s0, \$s1…
  - temporary variables are labeled as \$t0, \$t1…
- Previous example:

```
 add  $t0, $s1, $s2       #add  t0, g, h
add  $t1, $s3, $s4       #add t1,  i, j
sub  $s0, $t0, $t1       #sub  f,   t0, t1
```

University of Cape Town Josiah Chavula - 2020 Slide 15

Memory Operations

- As we can only work on registers, we need operations to move values between registers and main memory
- Key commands:
  - Load word - lw \$t0, address
    - Specifies register (\$t0) in which to store value that'll be read from memory.
  - Store word - sw \$t0, address
    - Takes value in specified register and writes it to a memory location
  - address operand is a second register where we're getting data from/writing data to



Memory instruction format

- The format of a load instruction:

```
lw   $t0,  8($t3)
```

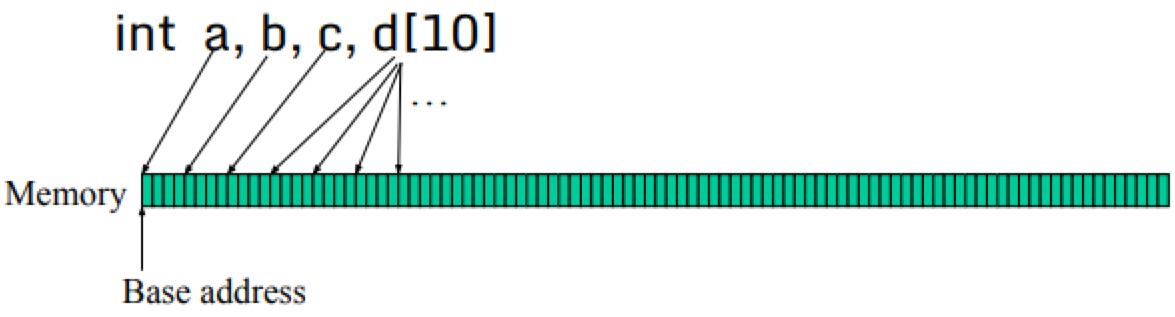
destination register
source address
a constant that is added to the register in brackets

University of Cape Town Josiah Chavula - 2020 Slide 17

- Constant used to calculate byte distance from a *base address*
    - Constant for nth value is (n-1)*4

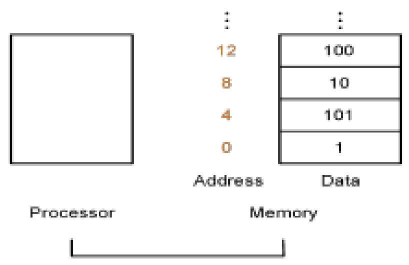
ADDRESS

- Computer memory is a single-dimension array
- Compiler keeps a look-up table mapping each variable to an address in this array
    - Keeps pointer to location/byte number where variable begins in memory.
- Memory addressing is at byte lvl (8 bits)
    - Each variable takes 4 bytes/32 bits.
    - Consecutive memory address are 4 bytes apart.
- In MIPS, register that contains address is in between []

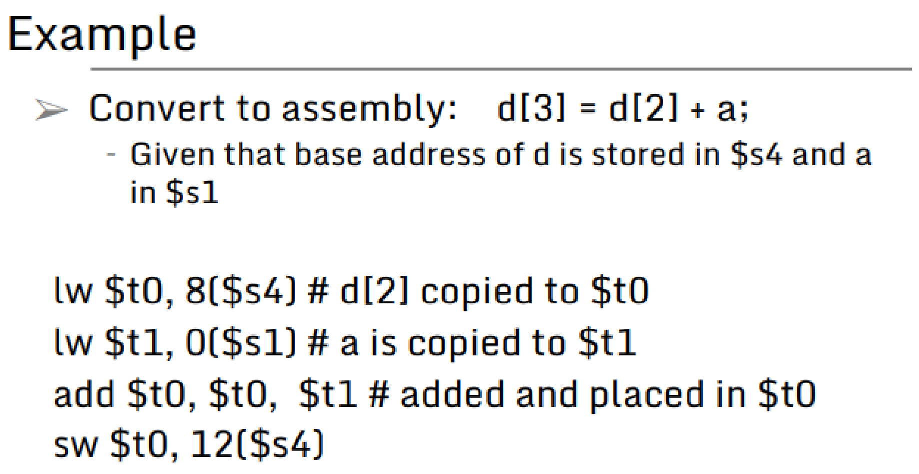


WORDS & BYTES

- NB: we’re working with 32-bit architecture
- Despite dealing with words, addressing is at the byte level.
    - Consecutive addresses are therefore a byte apart
- Addresses of consecutive words differ by 4
    - Each variable takes 32 bits (4bytes)



Example



Example

➢ Convert to assembly: d[3] = d[2] + a;
- Given that base address of d is stored in $s4 and a in $s1

```
lw $t0, 8($s4) # d[2] copied to $t0
lw $t1, 0($s1) # a is copied to $t1
add $t0, $t0,  $t1 # added and placed in $t0
sw $t0, 12($s4)
```

- a is already stored in register
- Need to get d[2] from memory into register – use load word (lw)
  - lw $t0, 8($s4) # d[2] copied to $t0
    - 8($s4) – [2] means it's stored in the 3rd position of the array
    - Each element of the array takes up 4 bytes
- Have to load value a into a temporary variable too
  - lw $t1, 0($s1) # a is copied to $t1
    - Because **a** is a constant, not an array
- add $t0, $t0, $t1 # added and placed in $t0
- sw $t0, 12($s4)
  - Have to take final value and place it back into memory
  - Want to add to 4th position in the array (index 3), so (4-1)*4 = 12, hence why 12 is constant

CONSTANTS

- You can add constants to values – values that aren't stored in a location in memory, they're part of the instruction
- This requires the use of an 'immediate' operator – add**i**
- One input must be a constant
  - E.g. addi $s0, $s0, 12

LABELS

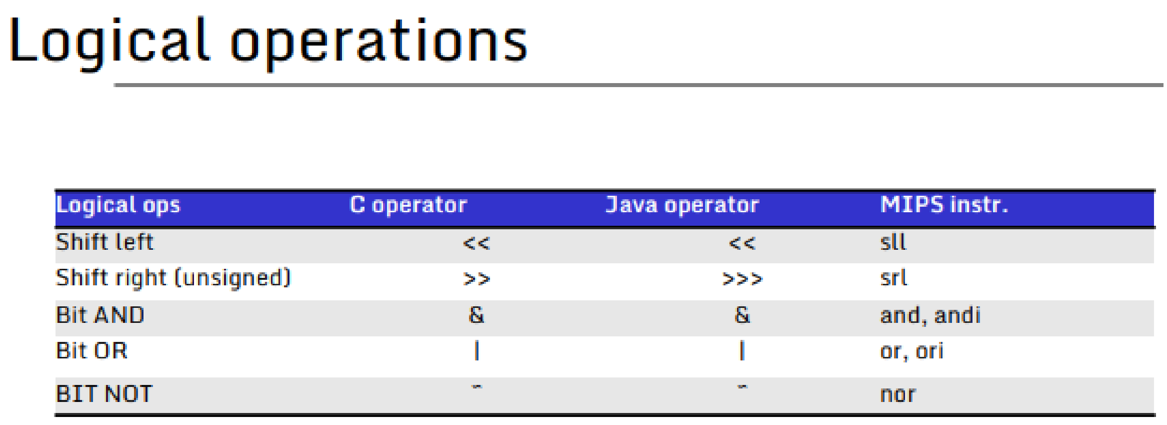
- MIPS assembly allows use of symbolic labels for pointing to the next instruction
- Usually, move sequentially from one instruction to the next (by incrementing Program counter, which points to the next instruction)

- This increment will be in sizes of 4 – each instruction made up of 32 bits, which is 4 bytes
- The jump takes place by adding an increment to the Program Counter
    - Overrides the default increment
- With labels, able to specify parts of logic where want to “jump to” given the output of an instruction.
    - When program compiled, labels will be translated to numeric values which dictate where to “jump” to
        - Instead of incrementing by 4, will add this numeric value to PC so that it will point to the next instruction that must be executed.
- Labels can be used with control instructions

Control instructions

- Conditional branch: beq r1, r2, L1 (**b**ranch if **eq**ual to)
    - Jump to instruction at address L1 if [r1] == [r2] i.e. if values in register 1 equals values in register 2, jump to part of program with label L1
    - Also **bne** (**b**ranch if **n**ot **e**qual to)
- Unconditional Branch – doesn’t depend on a result
    - E.g. after executes these instructions, jump to different part of program
    - E.g. j L1; jr $s0 (jump register – jumps to specific instruction pointed to by address in register).

Logical Operations

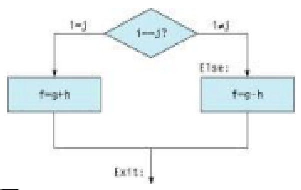

Logical operations

Logical ops	C operator	Java operator	MIPS instr.
Shift left	<<	<<	sll
Shift right (unsigned)	>>	>>>	srl
Bit AND	&	&	and, andi
Bit OR	|	|	or, ori
BIT NOT	~	~	nor

- Shift left (shift logical left): when have string of bits, shift left moves from lower to higher order
    - Insert zeroes to end of bit sequence – values shift
    - Basically, a multiplication of 2^(shift)
    - E.g. to multiply a number by 4, shift left by 2, since 2^2 is 4
- Shift right is a division by 2^(shift)

Convert

```
if (i==j)
  f = g+h;
else
  f = g-h;
```



```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else:
      sub $s0, $s1, $s2
Exit:
```

- Labels and branching can also be used in execution of loops

Loops

```
while (array[i] ==k)
   i+= 1;
```

i is $s3, k is $s5 and base addr of array is $s6

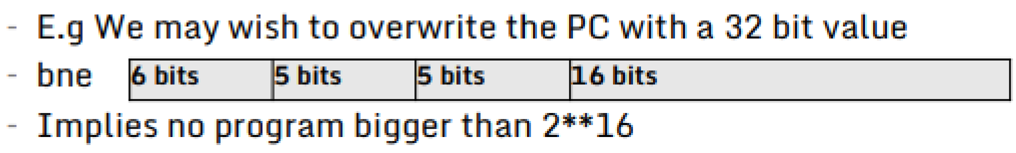
```
Loop: sll $t1, $s3, 2 #convert i to byte addr
         add $t1, $t1, $s6 #array element addr
         lw $t0, 0($t1)# deref array
         bne $t0, $s5, Exit # check loop end
         addi $s3, $s3, 1
         j Loop
Exit:
```

- Want to work out where to find element we're pointing to i.e where elements are located in terms of byte positions in memory
  - Have base address of array but need byte # of ith array item
  - Byte position of array item = index*4
    - Can multiply by 4 by doing a shift left with 2
    - sll $t1, $s3, 2 #convert i to byte addr
- Find element address by adding to base address
  - add $t1, $t1, $s6 #array element addr
- Then need to load value from memory into temporary register:
  - lw $t0, 0($t1)# deref array
    - Have zero(xx) to show that element is exactly where that address is pointing
- Next, need to perform computation:
  - bne $t0, $s5, Exit # check loop end
  - If value in $t0 (array[i]) is *not equal* to $s5 (k), then exit (since it's a while loop)
- If values are equal, then while loop continues
  - addi (add immediate) to increment value of i
  - addi $s3, $s3, 1

            - Take $s3, increment it by 1 and store result in $s3
- Conditional jump to go back to start of loop:
    - j Loop

32-BIT OPERANDS

- Problem: Instructions are 32 bits long and may refer to values that are also 32 bits long
    - i.e how do we take an address that's 32 bits and put it in an instruction that's *also* 32 bits?



- E.g We may wish to overwrite the PC with a 32 bit value
- bne

6 bits	5 bits	5 bits	16 bits

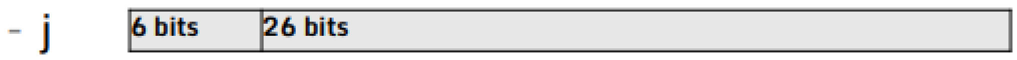
- Implies no program bigger than 2**16

    - for bne example – 6 bits is for cmd itself; set of 5 bits is for the register (1st 5 bits for 1st register, 2nd 5 bits for 2nd register).
        - 16 bits just for instruction and registers
        - Remaining 16 bits is used to indicate # of bytes need to move to get to nxt instruction
    - If only have these 16 bits for branching, means we only have 2^16 uniq positions (locations) that we can point to
        - Implication is that can't have prog bigger than 2^16 lines (won't be able to point to those locations otherwise)
- Good news is that most jumps take place within a few words of the current PC
    - PC (Prog Counter)-Relative Addressing
    - Jumps by words, not bytes – i.e. most jumps take place within a few words of current prog counter
        - Jump is not too far because it's normally relative to current location

BIGGER JUMPS

https://chortle.ccsu.edu/AssemblyTutorial/Chapter-17/ass17_5.html

- For bigger jumps, we can use the 'j' instruction which will give us a 26-bit addressable jump space
    - "jump" command (j) – only using 6 bits for cmd and remaining 26 can be used to pt to nxt address that we must go to.
    - More bits to rep diff locations in prog where we can jump to.



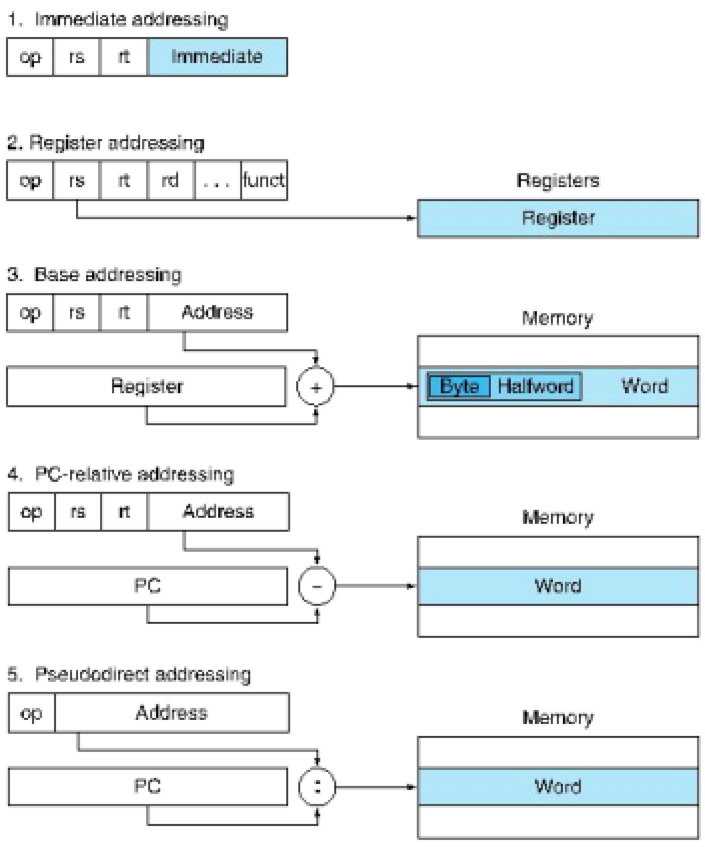
- j

6 bits	26 bits

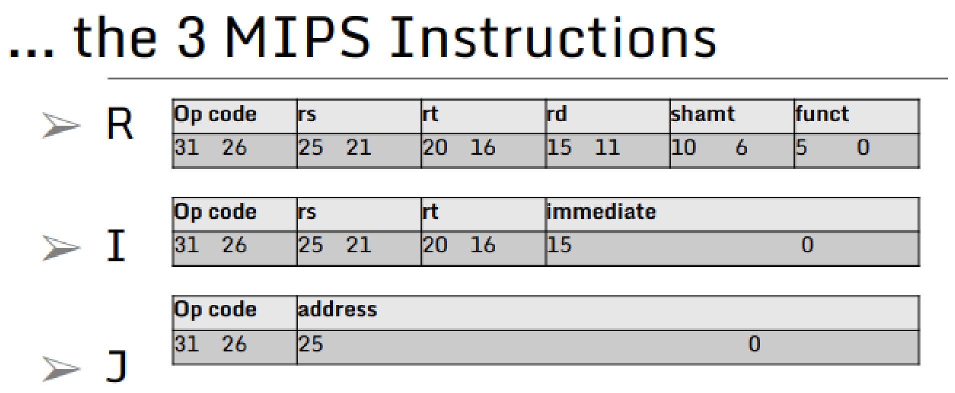
- Jump is also at the word level, so the 26 bits can address 2^28 bytes
    - When making jump, "jumping" 4 bytes in each subsequent jump – each point represents an address of 4 bytes
    - Locations are 2^26 bits, multiplied by 2^2 (which is 4), put together becomes 2^28
    - PC (prog counter) completed by the upper 4 bits of the current PC (4 + 28) – gives bigger space which we can address
    - Means the address space is in 256MB chunks
        - (ie 2^28 => 268,435,456 bytes => 256MB)

Addressing modes

- Register addressing
    - Operand is a register – values have to work on are in registers themselves
        - Just reading what is in reg and processing
- Base + offset addressing
    - Operand address is found by adding a register to a constant
    - E.g. used in lw instruction – source is given by constant(base address)
- Immediate
    - Operand is contained in instruction as a constant
    - E.g. addi reg,reg, **some value**
- PC-relative
    - Program counter
    - Final address where have to fetch nxt instruction is sum of PC and a constant
    - Normally used in branch instruction
- Pseudodirect Addressing
    - Point to specific mem location where have to fetch next instruction
    - Given in jump i.e. j
    - Top 4 prog counter bits concatenated to 28 bits of Jump instruction
        - (26 bits from instruction shifted left 2 bits)
    - 26 bits rep memory/address location where have to fetch next instruction
        - 26 bits are word location/address – to find specific byte address, multiply value of 26 bits by 4 (shift left by 2)
        - Remember, when finding mem address, counting in bytes, but addresses given in instruction are word addresses – counting in groups of 32
- Prog Counter relative vs Pseudodirect
    - With Prog Counter addressing, branch using what is in prog counter and # of steps/words have to jump
    - Pseudodirect – given actual mem address where have to find next instruction.



- 1. Immediate – val is part of instruction
- 2. Register – registers listed as part of instruction
- 3. Base addressing – address itself given by val in some register with constant added to it to find final mem address
- 4. PC relative – e.g. in branch, given numeric val which indicates how many words need to jump, add this to current prog counter to find address in memory (where to need nxt instruction)
- 5. Pseudodirect – have actual mem address and concat val with prog counter bits to find actual address in mem (get val from there)



... the 3 MIPS Instructions

➢ R

Op code	rs	rt	rd	shamt	funct
31 26	25 21	20 16	15 11	10 6	5 0

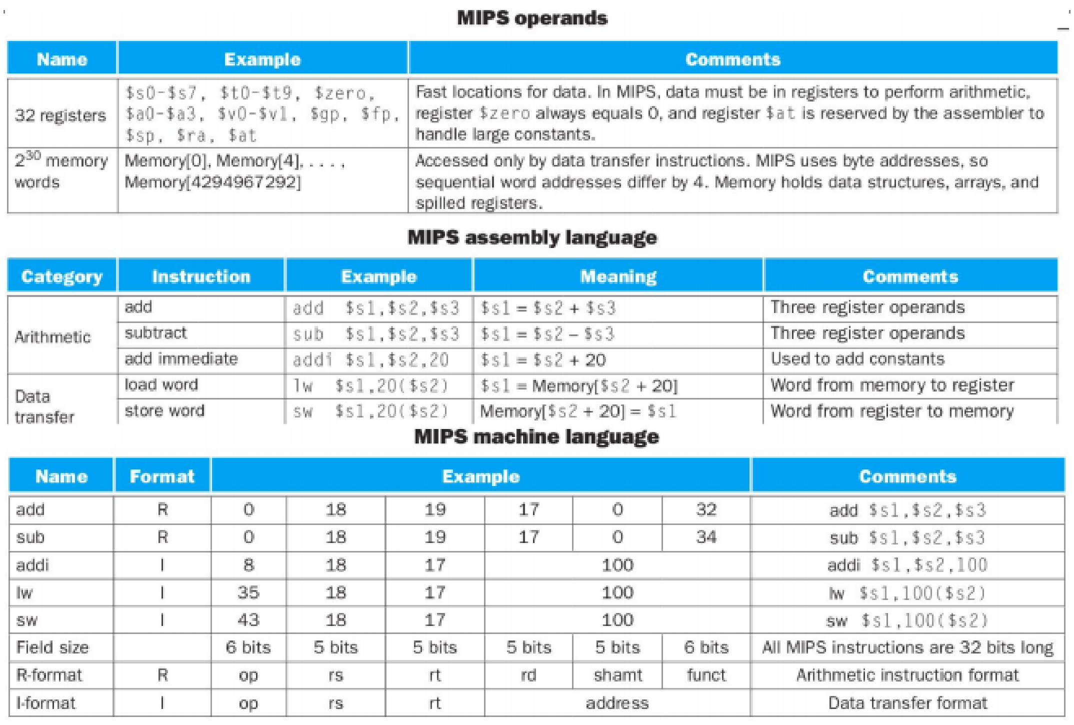
➢ I

Op code	rs	rt	immediate
31 26	25 21	20 16	15 0

➢ J

Op code	address
31 26	25 0

1. R:
   a. Register – instruction made up of regs and operate on registers
   b. E.g. add reg,reg,reg
2. I:
   a. Immediate – an immediate val
   b. E.g. addi reg,reg, value
   c. Also used in branching
      i. E.g. branching based on comparison of two register values and next to it, have val which indicates # of steps to jump forward/backward to get to nxt instruction.
3. J:
   a. Jumping – jumping to specific mem location
   b. Have given 26-bit address that work on, multiply by 4, then add to top 4 bits of prog counter and find location in mem to jump to



MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by 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 word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1 = $s2 + 20`	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1 = Memory[$s2 + 20]`	Word from memory to register
	store word	`sw $s1,20($s2)`	`Memory[$s2 + 20] = $s1`	Word from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	`add $s1,$s2,$s3`
sub	R	0	18	19	17	0	34	`sub $s1,$s2,$s3`
addi	I	8	18	17	100			`addi $s1,$s2,100`
lw	I	35	18	17	100			`lw $s1,100($s2)`
sw	I	43	18	17	100			`sw $s1,100($s2)`
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Processor

Response Time and Throughput

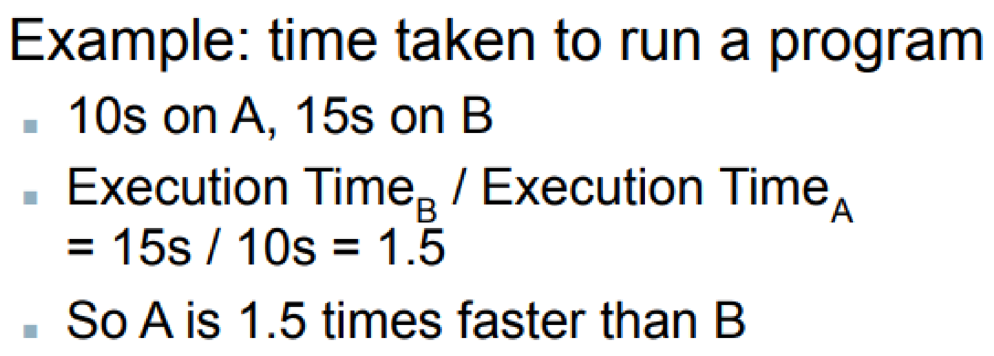
- Response time
  - How long it (computer) takes to do a task
  - This time includes time computer is accessing disk to read info from memory and performing input and output activities
  - "How long the user waits until their task is computed"
- Throughput
  - Total work done per unit time
  - How many tasks you can complete in a given time?
- How are response time and throughput affected by:
  - Replacing the processor with a faster version?
  - Adding more processors?
  - Reduces response time and increases throughput
- Focus: response time

Relative Performance

- Performance is reciprocal of response time
  - Performance = 1/Execution Time
  - To maximise performance, need to minimize response time – reduced execution time == greater performance
- A ratio:
  - "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

  - Can either compare based on performance or execution/response time



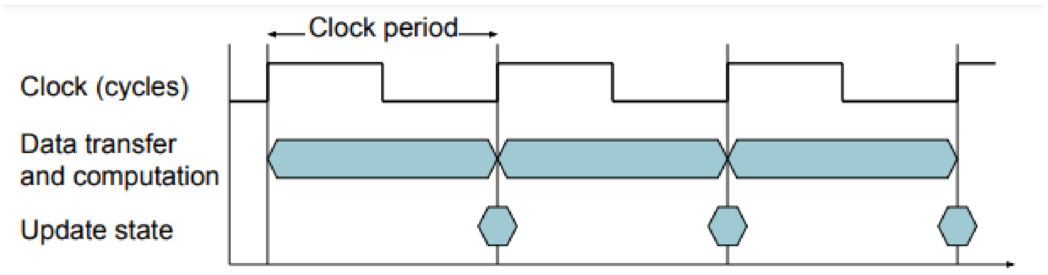
Example: time taken to run a program
- 10s on A, 15s on B
- Execution Time_B / Execution Time_A = 15s / 10s = 1.5
- So A is 1.5 times faster than B

MEASURING EXECUTION TIME

- Elapsed time

  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares (other programs)
  - Comprises user CPU time and system CPU time
  - Different programs are affected differently by CPU and system performance

CPU Clocking



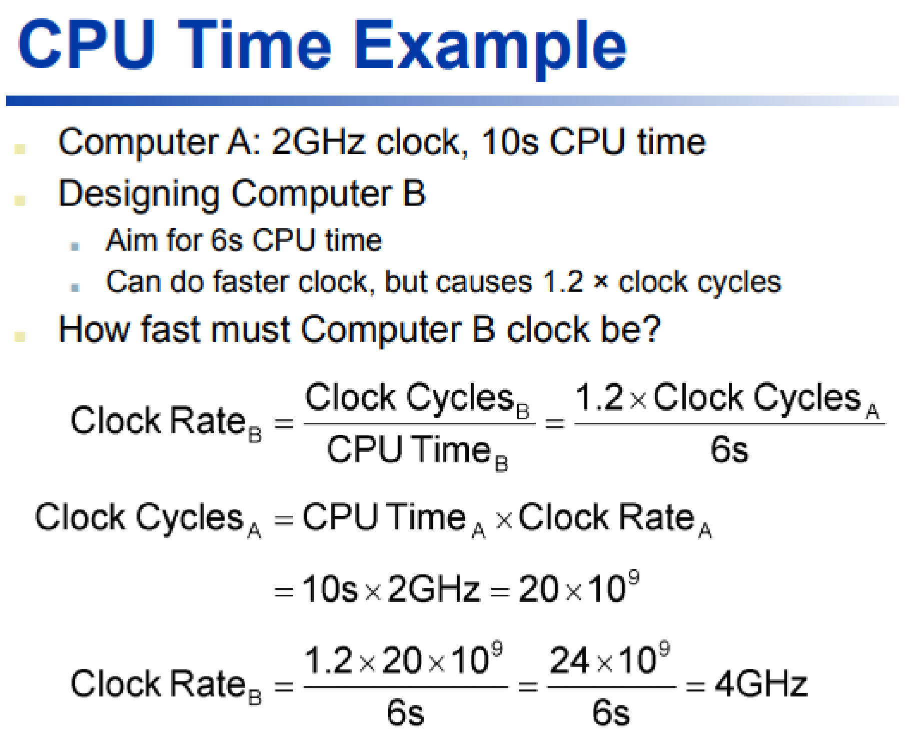
- Clock determines when *events* take place in hardware – at discrete time intervals
  - Operation of digital hardware governed by a constant-rate clock
- E.g. Data transfer and computation take place during clock cycle:
  - By the end of clock cycle, process ensures that with that task, state has been saved.
  - When go into next state, able to determine what value was when clock cycle was ending
- Values are changed only **during** clock cycle – not towards the end.
- Clock period/cycle time: duration of a clock cycle
  - e.g., 250ps (picoseconds) = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - Clock rate is inverse of clock period:
    - # of clock cycles that can be completed per unit time
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU TIME

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- CPU time used to compare CPUS
- CPU time: how much time CPU takes to compute task
- Performance improved by:
  - Reducing number of clock cycles that are required to complete task
  - Increasing clock rate – having more clock cycles within a second (more tasks completed within amount of time)

- Design difficult - Hardware designer must often trade off clock rate against cycle count
  - Increased clock rate makes cycles shorter



CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10s \times 2GHz = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4GHz$$

- For CPU time example:
  - 10s x 2Ghz = 10s x 2s X 10^9
- Clock rate improvement not necessarily directly proportional to performance improvement
  - In example, doubled the clock rate but only 4 seconds faster

INSTRUCTION COUNT AND CPI

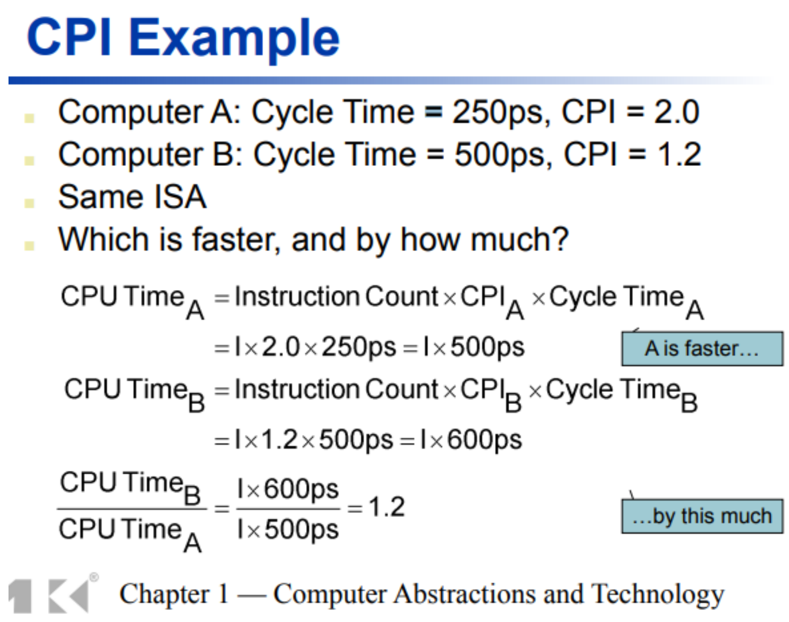
$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA (instruction set architecture) and compiler
- ISA: instruction set architecture
  - Also called architecture.
  - An abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory access, I/O, and so on
- Clock cycles per instruction (CPI): average cycles per instruction

    - i.e. how many clock cycles needed on avg for each instruction in a prog – NB that # of clock cycles needed by instruction can vary between instructions
    - Determined by CPU hardware
    - If different instructions have different CPI
        - Average CPI affected by instruction mix



CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$
\begin{aligned}
\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\
&= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \\
\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\
&= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps} \\
\frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2
\end{aligned}
$$

A is faster…

…by this much

Chapter 1 — Computer Abstractions and Technology

- For CPI example:
    - Let ISA = I

CPI in More Detail

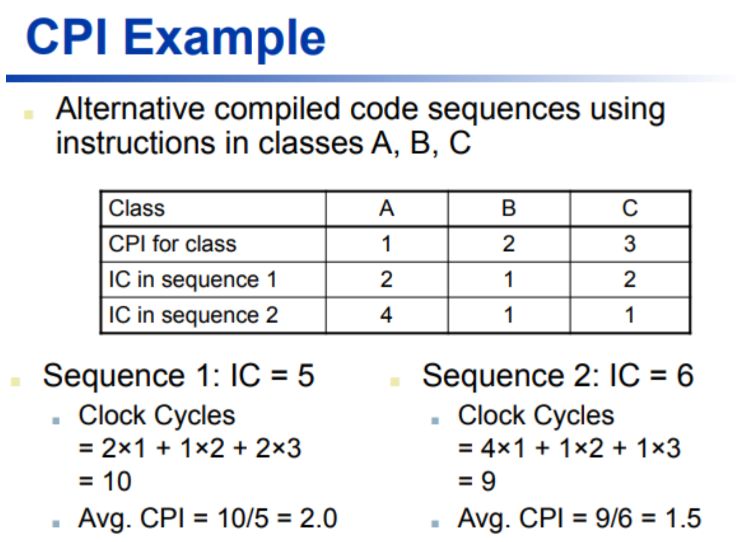
- Different classes of instructions can have different CPIs – need a way of computing them all together.
- If different instruction classes take different numbers of cycles:

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI:

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

    - Weighted avg CPI is summation of each class' CPI x relative frequencies
        - Result is a ratio



CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

  - Can have different sequences and depending on sequences of instruction, can end up with different instruction counts.
  - 5 = 2+1+2 (IC in sequence 1)
  - 6 = 4+1+1 (IC in sequence 2)
  - Seems like seq 1 is better than seq 2 since it has fewer instructions (5 < 6)
    - But actually, seq 2 is faster since it has less CPU clock cycles than seq 1
    - Each instruction takes less clock cycles to complete than seq 1

Performance Summary

Some extra questions with solutions: https://www.d.umn.edu/~gshute/arch/performance-equation.xhtml#:~:text=Clock%20time%20(CT)%20is%20the,cycle%20time%20of%200.25%20ns.

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- CPU Time:
  - Key elements that affect performance are instruction count, clock rate and clock cycles per instruction
    - Changing any one of these affects performance.
- Performance depends on
  - Algorithm: affects IC, possibly CPI
    - Algo determines src program instructions
      - Fewer prog instructions = better performance
    - Algo with simpler instructions = fewer clock cycles per instruction => better performance
  - Programming language: affects IC, CPI

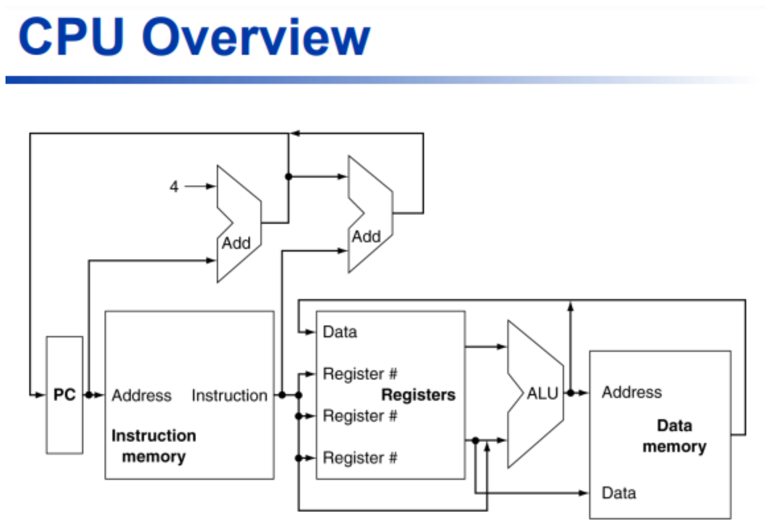
        - Statements in lang are translated to processor instructions – determines instruction count (IC)
        - Prog lang used also effects CPI – one with high lvl abstractions and/ complex data structures leads to execution of more cycles for each instruction.
    - Compiler: affects IC, CPI
    - ISA: affects IC, CPI, Tc
        - MIPS gives certain # of instructions per prog and clock cycles per instruction
        - RISC gives fewer clock cycles per instruction than complex architecture.

CPU

Instruction Execution

- Basic CPU: Math, Memory access, Branch/Jump
- PC → (points to) instruction memory, (where we) fetch instruction
- Register numbers → register file, read registers
    - Register file: collection of registers
    - Each instruction has number which → specific reg in reg file
        - Used when reading values from/writing values to reg
- Depending on instruction class:
    - Use ALU (Arithmetic Logic Unit) to calculate
        - Arithmetic result
        - Memory address for load/store
            - Remember with memory accessing - have offset value that will be added to address specified in reg
        - Branch target address
    - Access data memory for load/store
    - PC ← target address or PC + 4 (increments automatically since each instruction takes 4 bytes in memory)

CPU



1. Program Counter (PC): have specific register that contains value for PC
2. Instruction Memory:
    a. Architecture separates instruction memory and data memory (according to Harvard architecture – a contrast to Von Neumann)
3. Register file:
    a. Registers each with a number
4. Arithmetic Logic Unit (ALU):
    a. Used for arithmetic opns (adding) and memory access (ALU computes address we read from memory)

    b. Pass values to ALU
5. Add unit (leftmost):
    a. Initial and default
    b. Does PC increments – gets PC val, adds 4 to it and writes it back to PC
6. Add unit (rightmost):
    a. Used for when computing branch/doing a jump – computes address we want to go to

EXAMPLES

Load Instruction

- Doing basic **load** instruction
- Start by **fetching instruction** – use clock cycles
    - When cycle starts, begin to fetch instruction in address specified by program counter
    - Value of program counter then incremented by 4 – goes into initial Add unit
- Instruction must then be **decoded** – depending on type of instruction, operands for it will be read
    - For load instruction, have 2 operands – the reg to write values to and address in memory where we read values from.
    - Take instruction, go into register file to 'activate' reg that result will be written to (set it to be ready for writing)
    - Other values from instruction – constant value connects to ALU (goes *around* register file) and second value read from register will exit from register file and also go into ALU.
- **Compute addition** and find final value – the memory address where want to read something
- Memory address sent to data memory – get result from data memory and goes back to register file as data and is written into register in reg file

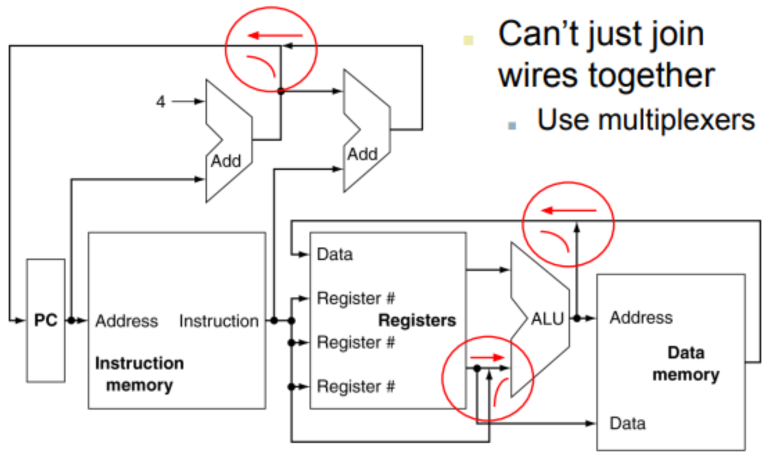
Add Instruction (R type)

- Start cycle with instruction fetch at beginning of clock cycle
    - Same process as previous example
- Instruction decoded and read
    - For R type instruction, have 3 operands (registers) indicated in instruction
        - Have 2 src regs – get values from these for opn
    - Point to reg file – where have 2 regs that we get values from
        - These values go straight to ALU
    - Destination reg is made ready to receive data
- Values are computed (result of addition) and sent back to reg file as data
    - Data is written to destination register
- Nothing is written to/read from memory and only use 1st Add unit

CPU OVERVIEW

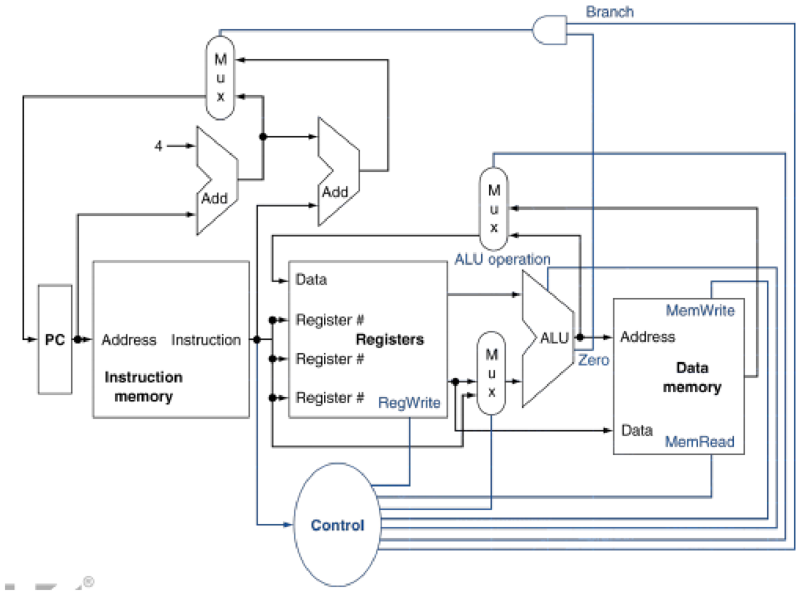
- Need to be able to explain functions of all elements at high lvl
- What are the roles of the Add units?
- Explain the inputs to the data memory unit
- Explain the inputs to the ALU
- Explain the inputs to the register unit

MULTIPLEXERS

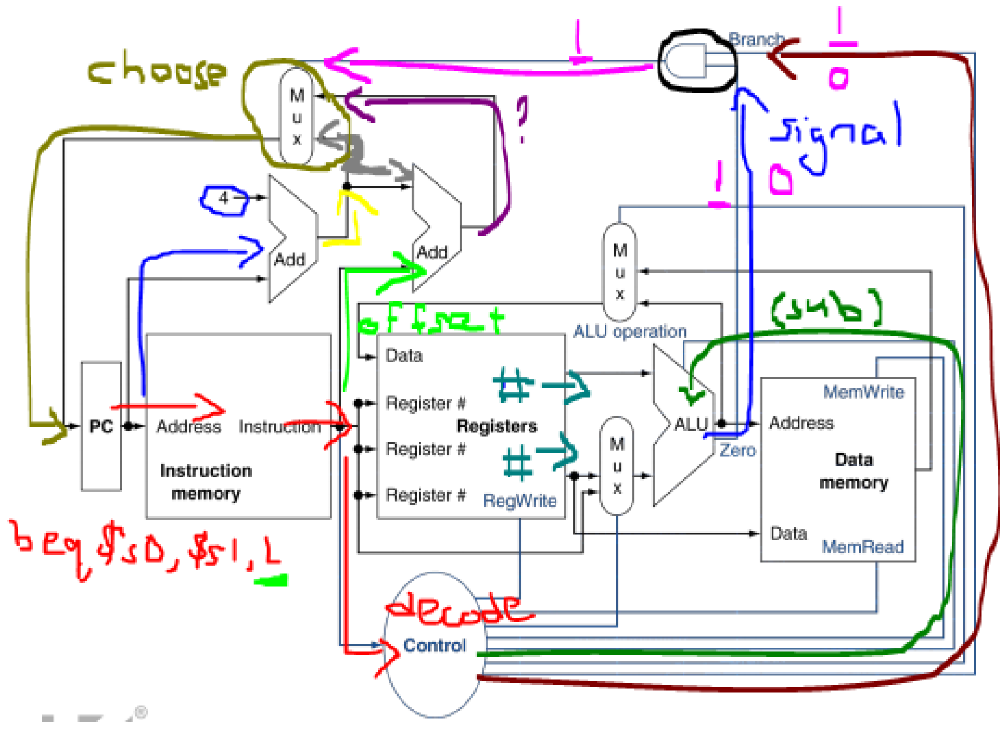


- Can't just join together – lines must be joined in such a way that we can decide which signal can be allowed to pass through
    - E.g. for the Add units, sometimes need to determine which signal is allowed to go through to prog counter
- Multiplexer will be used to switch between multiple signals and to allow specific signals to go through, depending on op that's supposed to happen.

Control



- Need controller to decide which parts of CPU must operate.
    - Decodes the instruction and activates different parts to be ready to execute different parts of the instruction
- Once instruction read from instruction memory, a copy of it will go into Control (controller)
    - Instruction then decoded in a way such that different CPU elements will be activated to operate on that instruction
    - E.g. for load instruction, one of regs in reg file has to be activated for data to be written to it.



Walkthrough Example

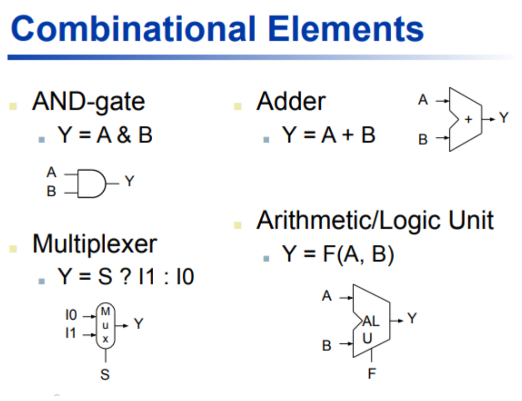
- Instruction: beq $s0,$s1,L (branch instruction)
- Prog counter points to nxt instruction we need to read from instruction memory
    - As soon as prog counter points to memory for instruction, increment PC by 4
- Instruction from memory then comes out
    - Copy gets sent to controller
    - Operands of instruction gets read from register file
- Control sees this is branch (I type instruction) and thus need to read reg values from reg file.
    - Send out two reg #s for regs that need to be read from reg file
    - Control informs ALU of what opn will happen here
- Since this is branch => ALU will need to compare those values
    - Control sends code for that to ALU – "since this is a branch, need to compare two values that are being sent to you" (tells signal which tells ALU what to do)
    - Opn is sub – is difference between them is zero => they're the same
- Zero signal comes out from ALU (i.e. whether difference is zero or not)
    - Control must also inform another element – helps us determine whether branch will be executed or not (branch condition has been met/not).
- If difference is zero => signal is positive (thus it's 1)
    - Assume +ve for this example – send +tve signal to multiplexor
    - Control also has +ve signal to indicate we're dealing with branch
- If both branching and zero signal are **true**, then branch condition holds, and branch takes place.
    - If zero signal was false, branching wouldn't have happened.
- Multiplexor:
    - Branching offset (value we must branch with) sent to second Add unit (L in code)
    - PC+4 that was computed earlier by first Add unit also sent to second Add unit
    - Add these two together – have new PC value that goes into multiplexor

- At the same time, the PC+4 that was computed by initial Add unit also sent to multiplexor
- Now have *two* PC values that multiplexor must choose from:
  - Will determine which values must be allowed to into PC
  - Does this based on signal from branching element – since in this example branching condition had been met, and signal is thus +ve
  - Consequently, multiplexor allows value from **second** Add unit to pass through and be written to PC

Logic Design Basics

- Info encoded in binary
  - Low voltage = 0, High voltage = 1
  - When trying to move data between elements, move info such that each bit moves on a specific wire
    - One wire per bit
  - Multi-bit data encoded on multi-wire buses
    - Move all bits at the same time
    - E.g. want to move 32 bits, then want to have 32 wires that will form a bus in order to move all bits in one go.
- Two main types of elements that form processor:
  - Combinational element
    - Operate on data – receive input as data
    - Output is a function of input
    - E.g. if two values sent in, and opn is addition, then result that comes out will be a function of that element
  - State (sequential) elements
    - Do not act on info – only receive it
    - Store information

Elements



Combinational Elements

- AND-gate
  - Y = A & B
- Adder
  - Y = A + B
- Multiplexer
  - Y = S ? I1 : I0
- Arithmetic/Logic Unit
  - Y = F(A, B)

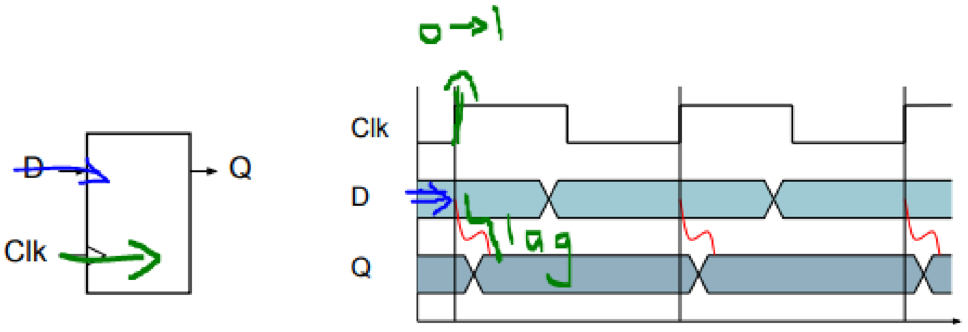
- AND gate: +ve only if boh inputs +ve
- Adder: adds values it receives
- Multiplexor: chooses one of inputs based on a 3rd signal S – this signal determines what input is allowed to exit as output
- ALU: function performed on input – output is function of input

    - Receives signal F that determines what type of computation it must perform on inputs

SEQUENTIAL ELEMENTS

Register

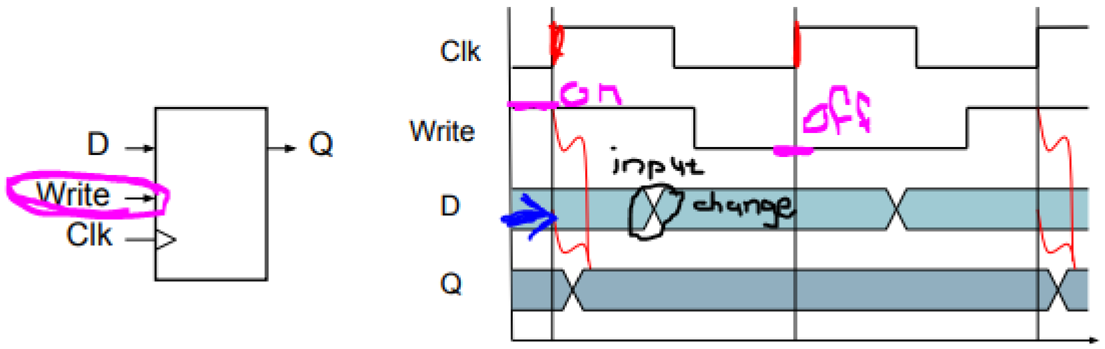
- Register: stores data in a circuit
    - It's simply a circuit that maintains some state and based on the state, can tell what value is in there
    - Uses a clock signal to determine when to update the stored value
        - Clock signal is the moment when a cycle begins
    - Edge-triggered: update their state when clock signal changes from 0 to 1



- Process:
    - Clock begins – changing from 0 to 1
        - At that point, clock comes in (clock moment)
    - Input comes in (new data) – must be written to element
    - During clocking time, that's when data in element will be updated
        - Thus, there's a moment of lag – unstable as it changes from 0 to 1
        - Once signal is stable, value in element will be updated to match input signal
    - Even if input signal changes again along the way, so long as this doesn't happen during edge triggering, value in element won't be changed.
        - As get to next edge of clock, then element will once again be updated.

Register with Write control

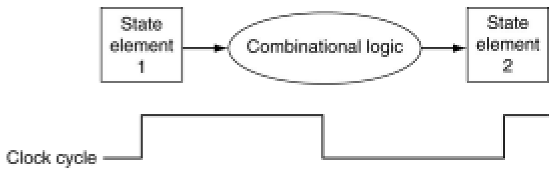
- Want more control in terms of updates
- Only updates on clock edge when write control input is 1
- Used when stored value is required later
- Write signal – this could be signal from Control
    - Goes to reg and tells it "need to write data that you receive on input"

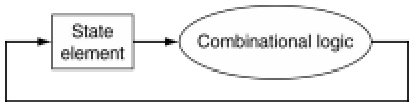


- Even though input changed, we ignore it since clock not on +ve edge
- When clock *is* on +ve edge but write signal is off, then do not update element

CLOCKING METHODOLOGY

- Combinational logic transforms data during clock cycles – they receive input at beginning of cycle, then have to work on data and be ready to write it into another state element at end of that cycle
    - Combination logic is not able to keep state – this means if clock cycle shifts to next one, then will have lost validity of data that is in combinational logic
    - Between clock edges
    - Input from state elements, output to state element
    - Longest delay (i.e longest action/task) determines clock period
- Actions are edge-triggered – writing happens as move *up* clock cycles

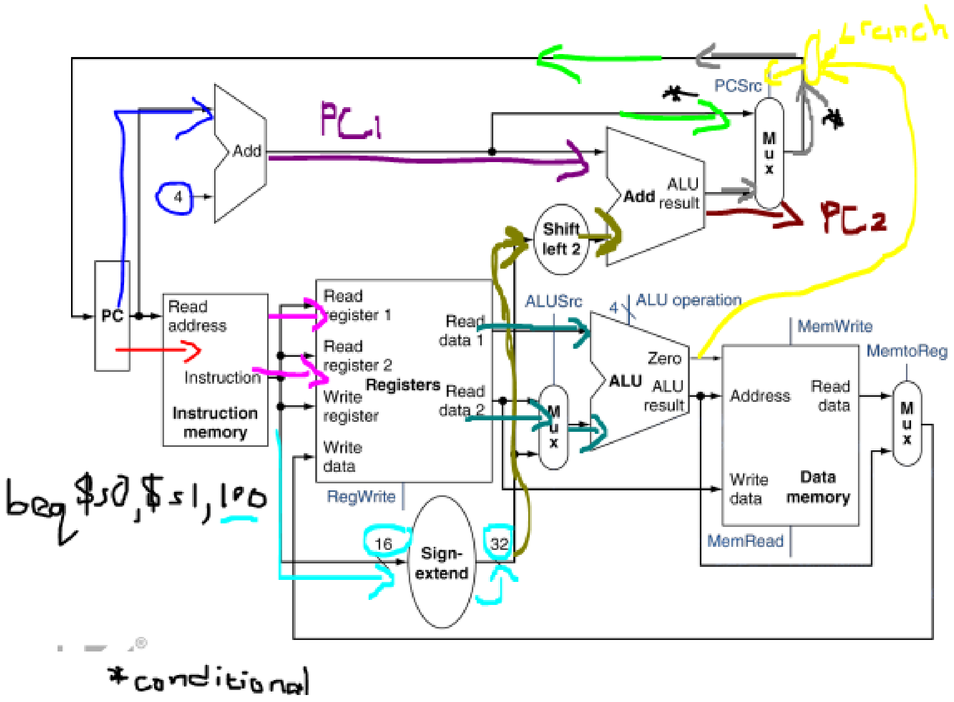




COMPOSING THE ELEMENTS

- First-cut data path does an instruction in one clock cycle
    - Each datapath element can only do one function at a time
        - E.g. if given an add, can only do one add at a time
    - Hence, we need separate instruction and data memories
        - Can simultaneously be reading something from instruction memory and within same clock cycle be reading/writing something from data memory
- Use multiplexers where alternate data sources are used for different instructions
    - E.g. have same ALU but it'll receive different inputs for different instructions – for add, may get input from registers, but input can be different for a branch

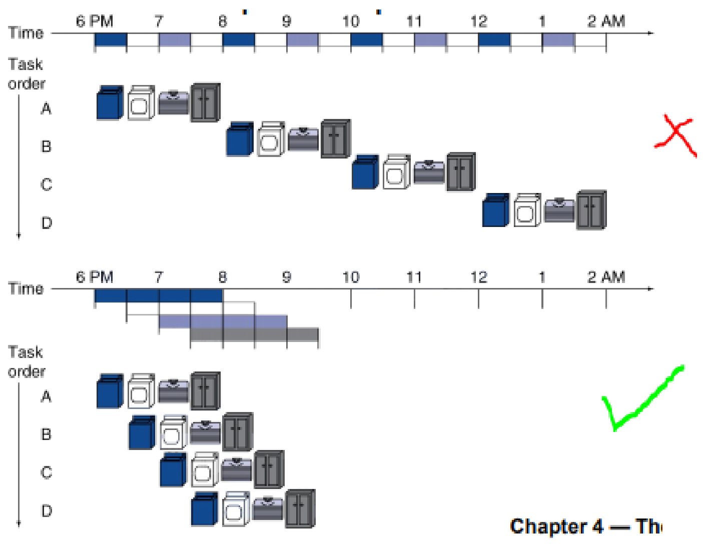
Full Datapath



Performance Issues

- Longest delay determines clock period – i.e longest instruction to compute determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
  - Inconvenient for processor
  - Violates design principle
    - Making the common case fast – load instruction may not be most common instruction, but it's going to delay most frequent instructions
- Improve performance by pipelining

Pipelining



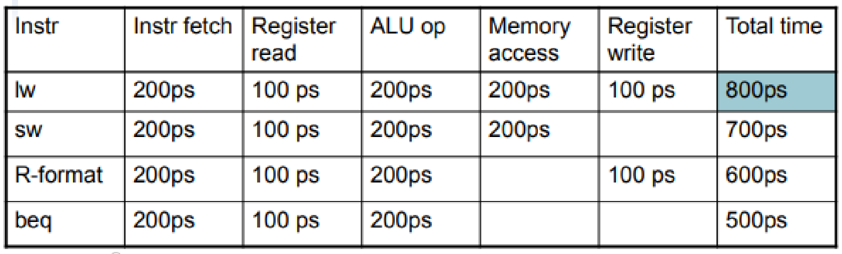
- Analogy
  - Pipelined laundry: overlapping execution
    - i.e execute multiple instructions simultaneously
    - "While I'm drying my laundry, someone else can use the washing machine at the same time"
  - Parallelism improves performance

MIPS Pipeline

- Five stages, one step per stage
  - IF: Instruction fetch from memory
  - ID: Instruction decode & read regs from reg file
    - Occurs simultaneously
  - EX: Execute operation or calculate address
  - MEM: Access memory operand
  - WB: Write result back to register

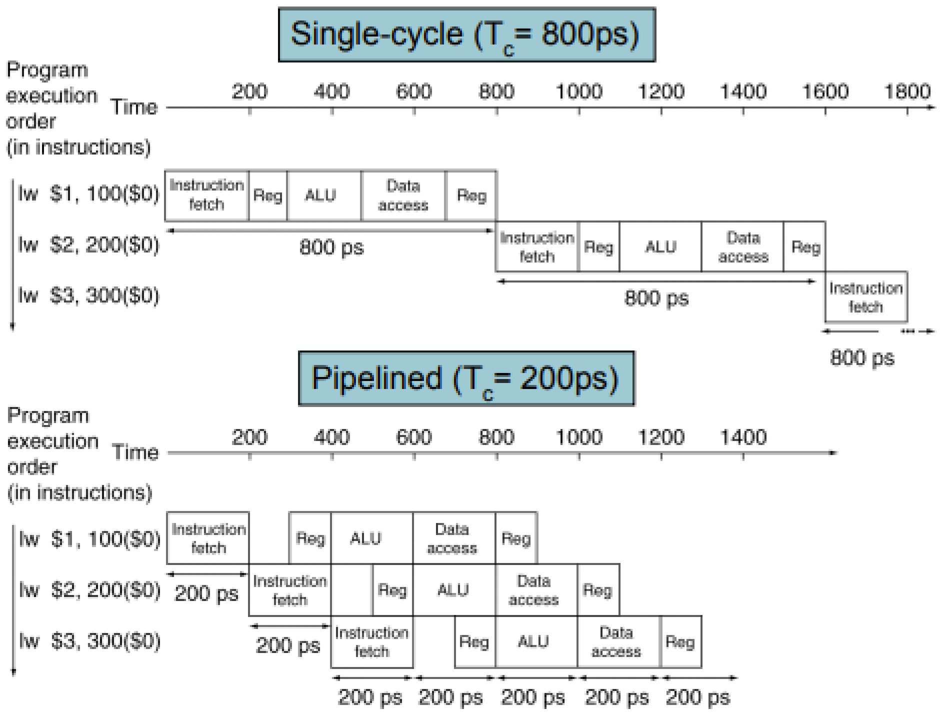
PIPELINE PERFORMANCE

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath
- For example, a single-cycle datapath:
  - lw takes 800ps



Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- Pipelined vs single cycle example:



- For single-cycle process, each instruction has to wait 800ps before can move on to next instruction
- For pipelined, only wait till first stage completed before starting (200ps)
  - Almost four-fold speedup

PIPELINE SPEEDUP

- If all stages are balanced (i.e., all take the same time)
  - Time between instructions$_{\text{pipelined}}$
    = Time between instructions$_{\text{nonpipelined}}$ /Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease – analogously, if it takes 2 hours for you to complete laundry, this won't change
  - Beginning instructions more frequently – means we can complete more instructions in a smaller amount of time

Pipelining and ISA Design

- MIPS ISA designed for pipelining

- All instructions are 32-bits
    - Easier to fetch and decode an instruction in one cycle – get the whole thing at once
    - c.f. x86 instruction: 1- to 17-byte instructions
        - Not a process that can be completed in one cycle, can only decode a part of instruction at a time
- Few and regular instruction formats
    - Can decode and read registers in one step
- Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
    - Memory access takes only one cycle
    - When trying to get something from memory, operands will be lined up next to each other, so can read them all at once

Hazards

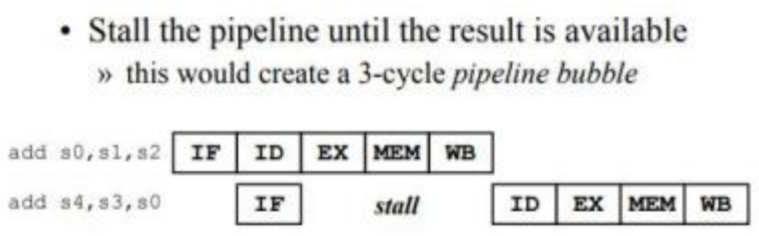
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards – planned instruction can't execute because h-ware doesn't support combo of instructions that we want to execute
    - A required resource is busy
        - E.g if we have an instruction that is trying to access resource that previous instruction still using
- Data/pipleline data hazard - planned instruction can't execute in its proper cycle because data needed to execute instruction is not yet available
    - Need to wait for previous instruction to complete its data read/write
- Control/branch hazard
    - Deciding on control action depends on previous instruction

STRUCTURE HAZARDS

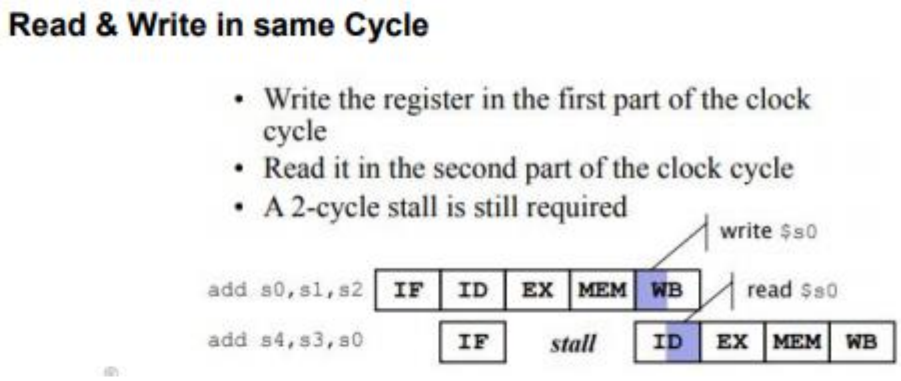
- Conflict for use of a resource
    - E.g. memory access – hence why need to separate data and instruction memory (to avoid this hazard)
- In MIPS pipeline with a single memory (i.e. instruction and data memory as one)
    - Load/store requires data access
    - Instruction fetch would have to stall for that cycle
        - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
    - Or separate instruction/data caches

DATA HAZARDS

- An instruction depends on completion of data access by a previous instruction



- Stall the pipeline until the result is available
  - » this would create a 3-cycle *pipeline bubble*



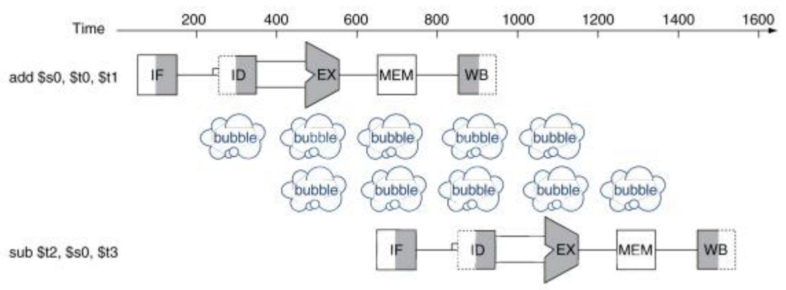
Read & Write in same Cycle

- Write the register in the first part of the clock cycle
- Read it in the second part of the clock cycle
- A 2-cycle stall is still required

- For example:
  - add s4, s3, s0 – need s0, which is result of previous instruction
    - s0 may not yet be ready – it's only ready when Wb (write back) happens
  - Need to *stall* to prevent error – have to wait 3 cycles before can get s0
- Processors reduce waiting time by trying to ensure w/r of regs happen in one cycle – i.e. in the same cycle
  - Now only need to wait for 2 cycles here

Stalling...

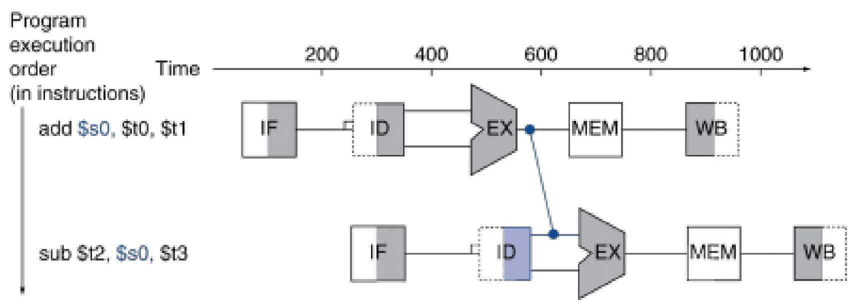
```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```



- Bubbles indicate we need to wait

Forwarding (aka Bypassing)

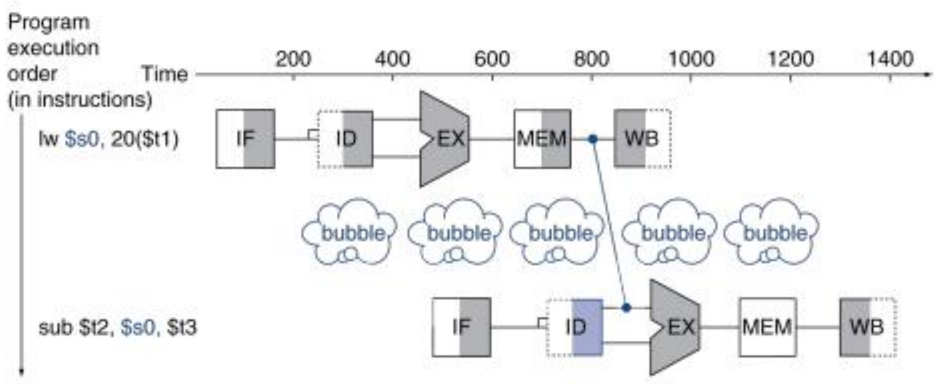
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
    - Processor must be designed such that can connect necessary components



- With forwarding, don't have to wait for anything

LOAD-USE DATA HAZARD

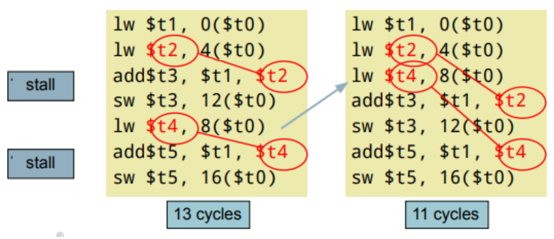
- Can't always avoid stalls by forwarding
    - If value not computed when needed
    - Can't forward backwards in time!



- In example: $s0 only available after memory access
    - Can only then be passed to execution line of next instruction

CODE SCHEDULING TO AVOID STALLS

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F; #array [B, E,F,A,C]
    - #array [B,E,F,A,C]



```
lw $t1, 0($t0)
lw $t2, 4($t0)
add$t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add$t5, $t1, $t4
sw $t5, 16($t0)
```

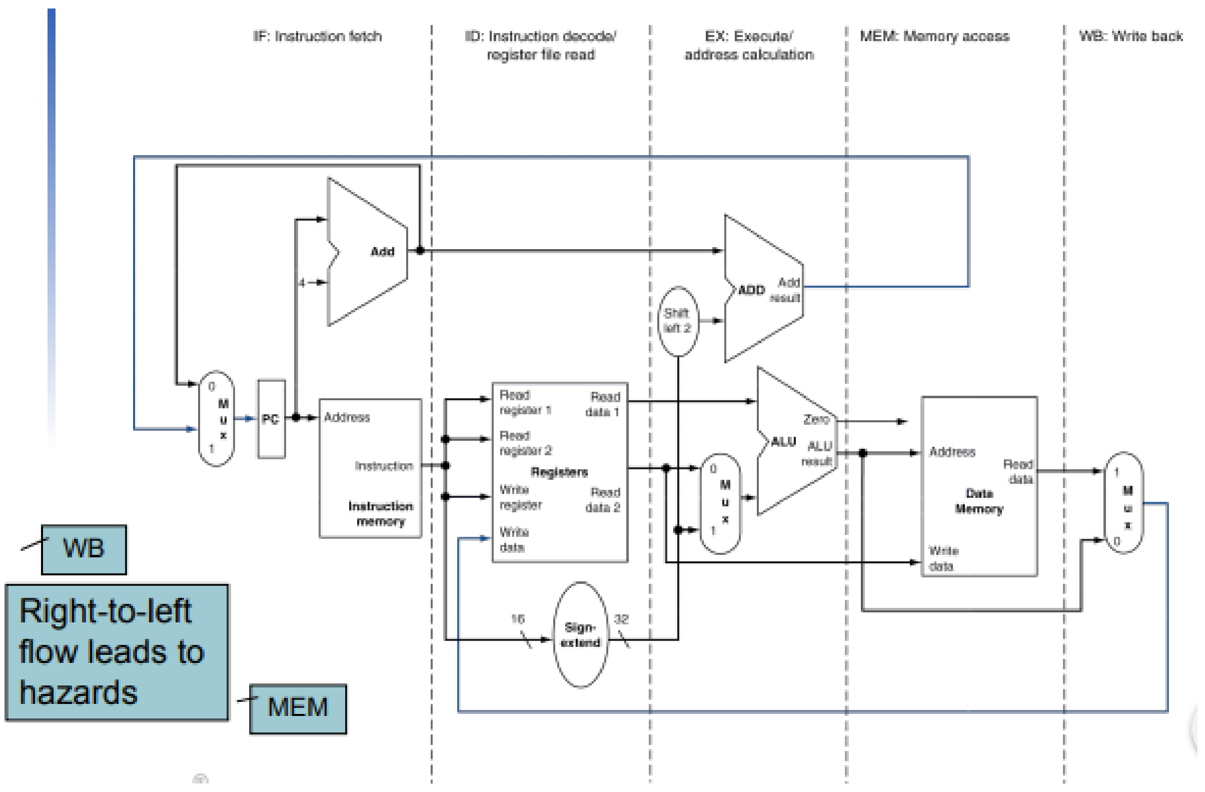
```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add$t3, $t1, $t2
sw $t3, 12($t0)
add$t5, $t1, $t4
sw $t5, 16($t0)
```

- Can re-order instructions that aren't dependent on each other, so that they follow each other
    - Can re-order in such a way that is minimizes stalling

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
    - Executes multiple instructions in parallel
    - Each instruction has the same latency
- Subject to hazards
    - Structure, data, control
- Instruction set design affects complexity of pipeline implementation
    - Regularity of instructions (always 32 bits, etc) enable pipelining

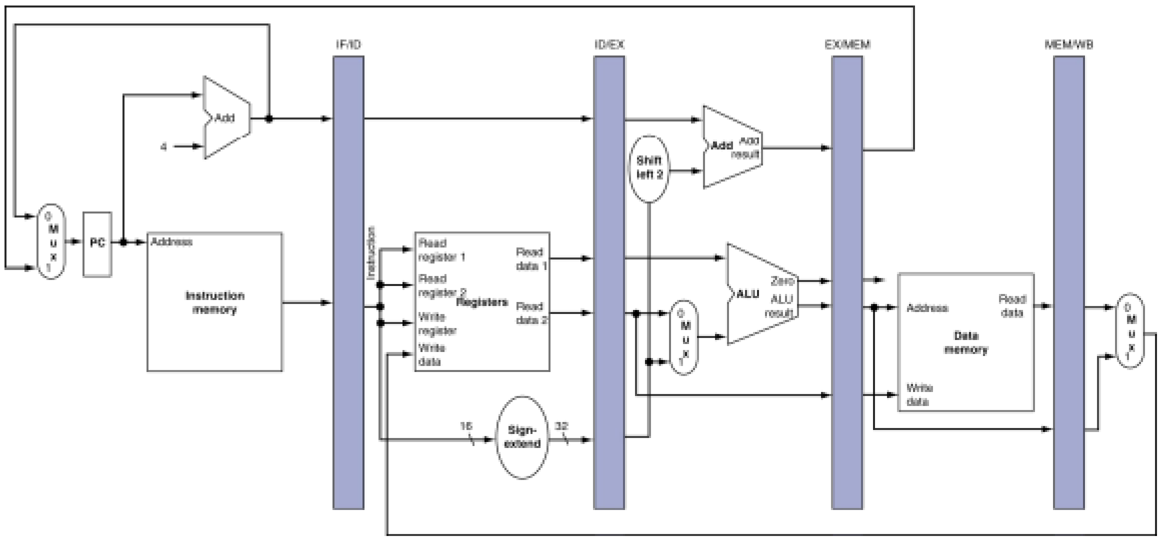
MIPS Pipelined Datapath



- Five stages in complete datapath
    - Single cycle datapath
- Allows up to 5 instructions to be executed in a clock cycle
- Each step in the instructions can be mapped onto datapath from right to left– exceptions are prog counter update and write-back step (writing to reg file)
- Right to left flow leads to hazards – want it to be left to right

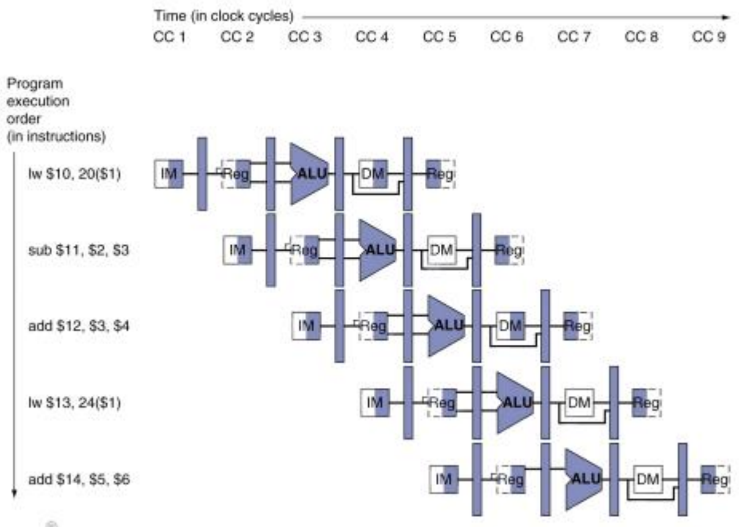
PIPELINE REGISTERS

- Need registers between stages
    - To hold information produced in previous cycle



- Recall – combinational elements don't keep state (can't store info beyond one cycle)
    - To have a staged data pipeline and be able to retain state, values must be saved in regs in between stages
        - Laundry analogy: need a basket to hold clothes before can move to next stage
    - Need "holding space/stage" for data as it moves between stages
- Have pipeline registers – inserted between stages of data pipeline
    - Labelled by stages they separate
    - Example: IF/ID is between instruction fetch and decode stages

Multi-Cycle Pipeline Diagram

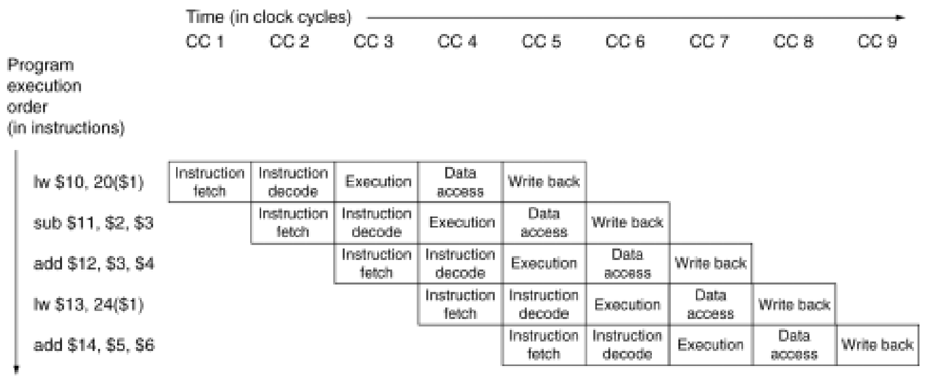


- Form showing resource usage
- To show what happens in pipeline execution – assume each instruction has own datapath
    - Place each datapath on a timeline to show their relationship
- Go from left to right
- Instructions listed in execution order

    - Each stage labelled by physical resource used in that stage
    - *IM* represents the instruction memory and the PC in the instruction fetch (IF) stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on.
    -
- Shade in each portion of datapath element according to the portion of each cycle when that element is in use
    - For example (image above): instruction memory only used during second half of that clock cycle, but ALU used during entire cycle so it's shaded in completely
    - Assume reg is written to in first half of cycle and is read during second half of the cycle
    - In diagram, reg file is in *two logical parts* – because reg may be accessed at two different times
        - During instruction decode (beginning)
        - During write back process (at a later stage)
        - Represented by dashed lines in unshaded half of reg file in ID stage - indicates that part does not yet exist
            - Same for WB stage – indicates that it isn't being read
    - Add regs between stages to hold data – portions of single path can be shared during instruction execution
    - Alternative explanation: To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, **when it is not being written**, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

MULTIPATH CLOCK PIPELINE

- Can also represent multipath clock pipeline
    - i.e. represent clock cycles in pipeline fashion without drawing elements
- Traditional form


Time (in clock cycles)

Program execution order (in instructions)

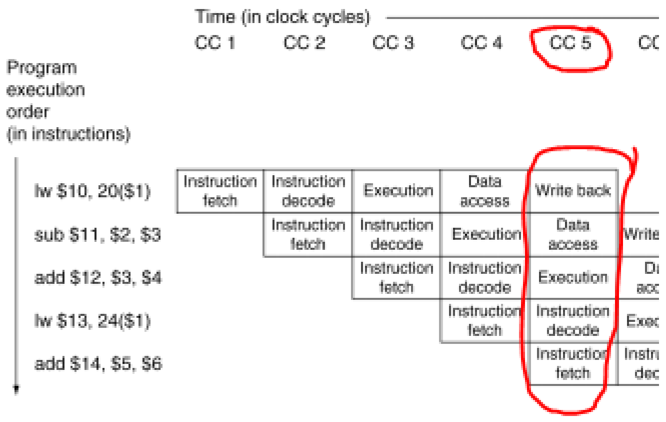
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

- Use names of stage happening at each step
    - List instructions in order from top to bottom

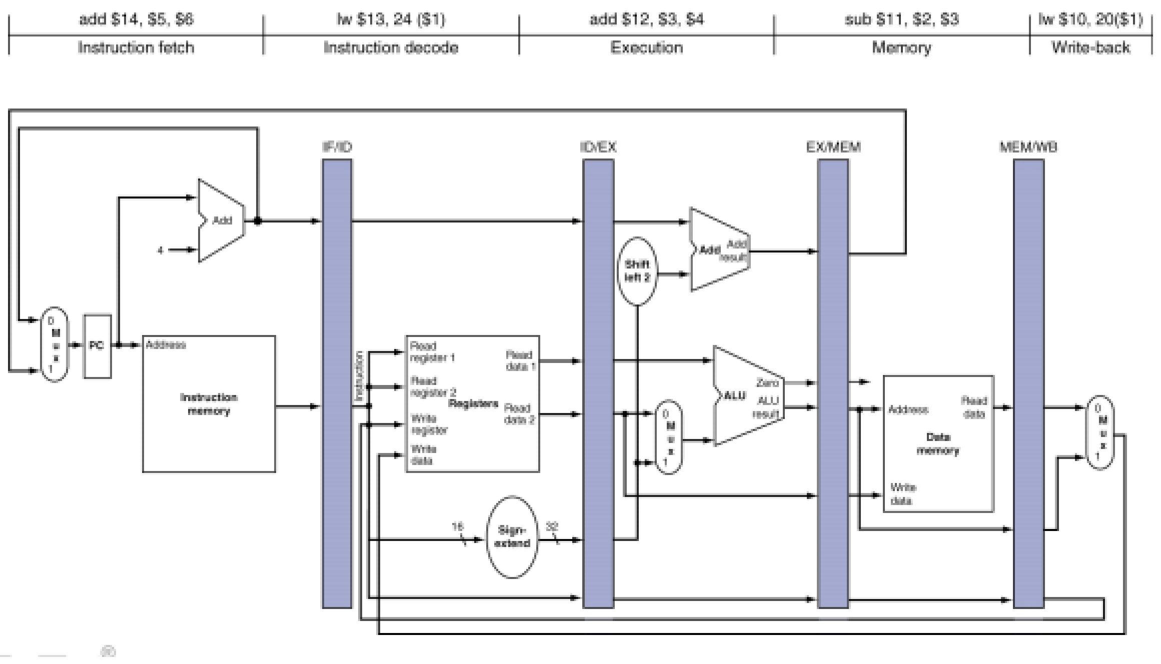
- Clock cycles go left to right

Single-Cycle Pipeline Diagram

- “Vertical slice” of traditional form multi -cycle pipeline diagram

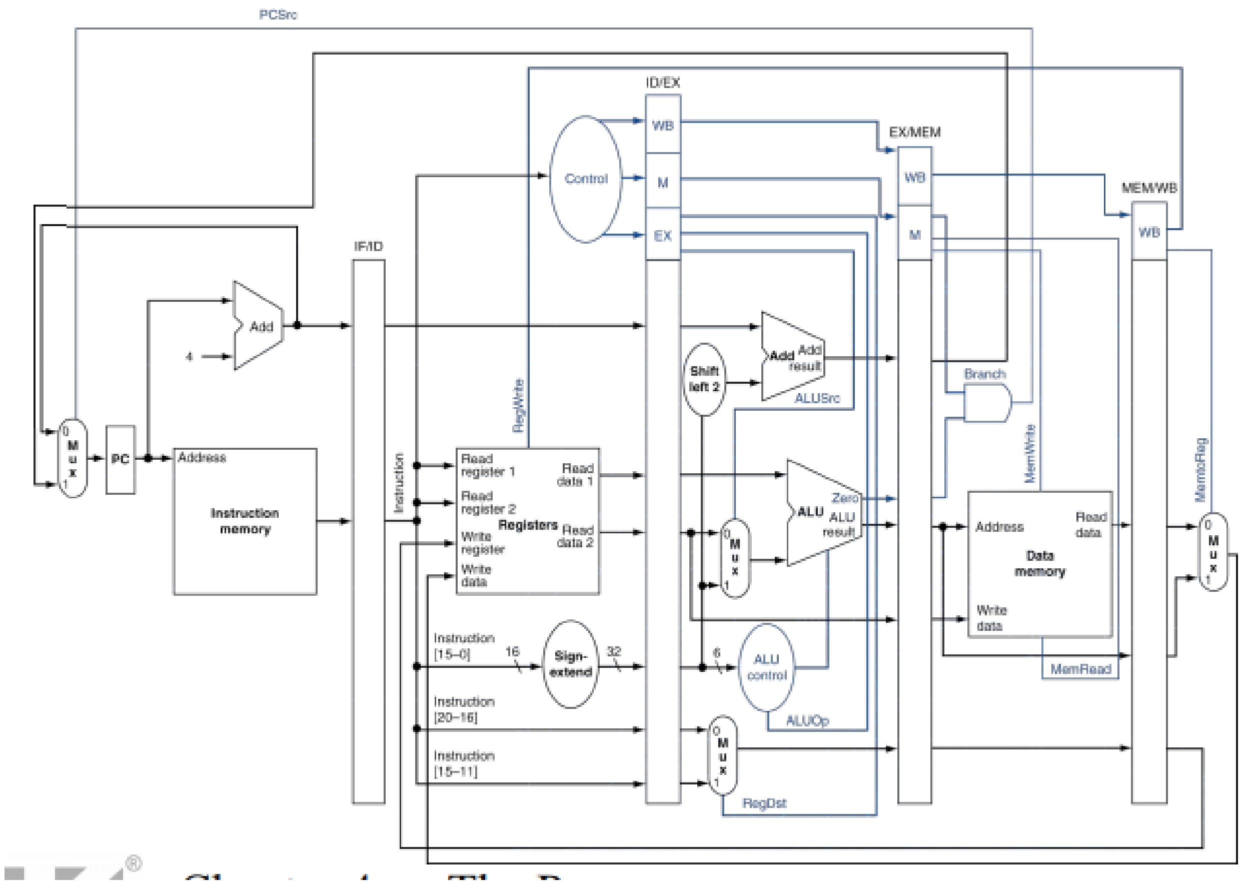


- State of pipeline in a given cycle
    - NB to read r to l



- For diagram, each instruction is a stage
    - Each stage thus has different instruction

Pipelined Control

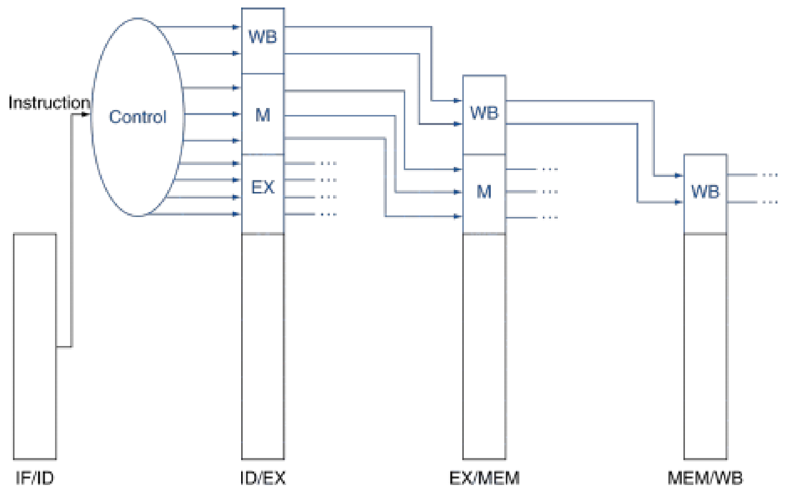


- Have to add control to pipeline – single cycle controls only are insufficient
    - Specific controls for pipelined architecture
- Key difference between single cycle and pipelined controls: control values are passed along though pipeline regs as each instruction moves through the stages
    - Control thus for each specific instruction must be passed at well
    - This is because at each stage elements are dealing with different instruction (as opposed to single cycle, where it was just the same instruction)
- Each controller is associated with component that is active in only a single, specific pipeline stage (depending on instruction that is moving at that stage)
- Can hence divide control lines into 5 groups according to pipeline stage
    - All controls that need to be executed can be placed in these groups
    - IF has control signals to read instruction memory and write prog counter
        - These same controls would appear in single cycle datapath
    - Next group: ID and reg read
        - Controls that allow r/w of reg in reg file and decoding instruction
    - Execution and address calc: signals for determining reg destination (where to send result) and opn/control of ALU opn – indicates to ALU what opn to work on
        - ALU source indicates which input signal must accept - depending on instruction
    - Memory access: setting the branch, memory r/w
    - Final stage is WB – control lines indicate whether write result of mem to reg
        - Or actual writing of reg to indicate to which reg we are writing in reg file

- These instructions/ controls are grouped according to these different stages
- To specify control for pipeline, only need to set control values during each pipeline stage
    - Control would have been read from beginning
        - As each instruction moves along those stages, different controls would be set (at each of those stages) depending on the different groups of control and where the instruction is at in the pipeline

CONTROL

- Control values for pipeline created during ID stage
    - Each instruction, as it gets decoded, has its control values calculated
    - Control element works out control signals
        - These signals are placed into ID/EX pipeline reg as instruction moves
            - Will be activated when instruction reaches next stage
- Some control lines for each stage are used – remaining gets passed to nxt pipeline stage
- Control signals derived from instruction
    - As in single-cycle implementation

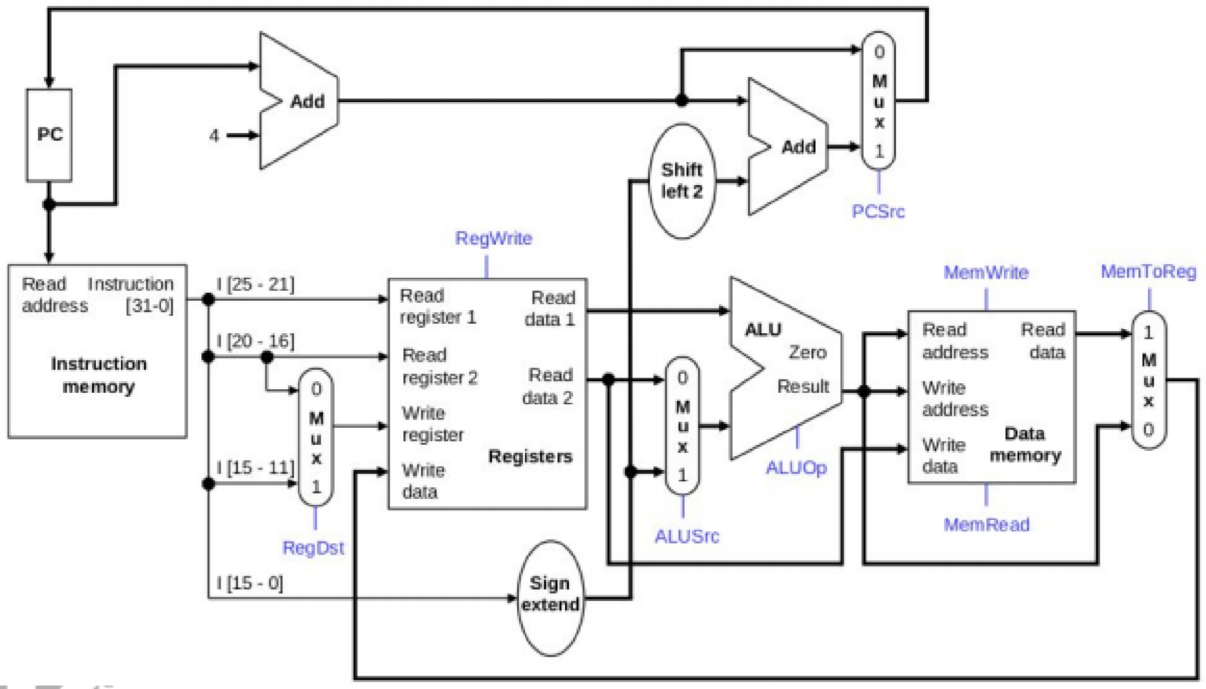


Recap

- Examine previous data pipelines i.t.o perspective of different types on instructions

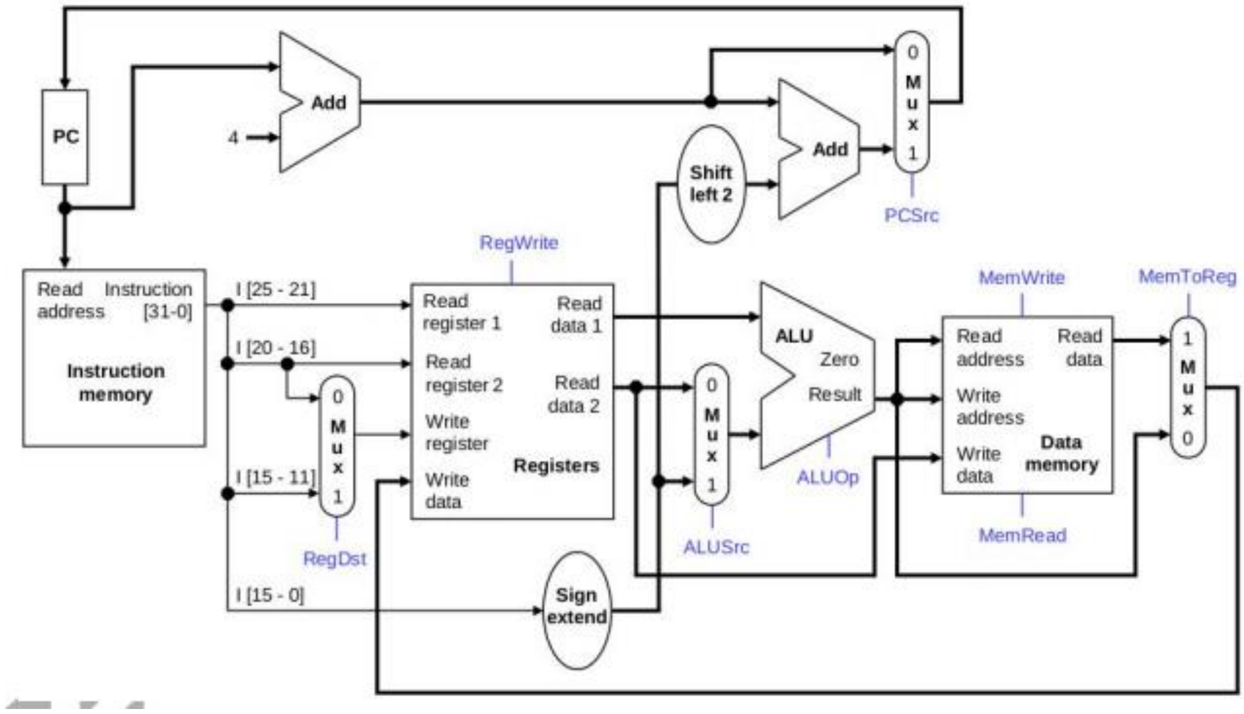
Recap: R-type instruction path

- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction’s ‘func’ field.



- Control codes sent to different elements
    - At each stage, as instruction is moving, see e.g. reg r/destination signals , ALU source or ALU opn code, being set.

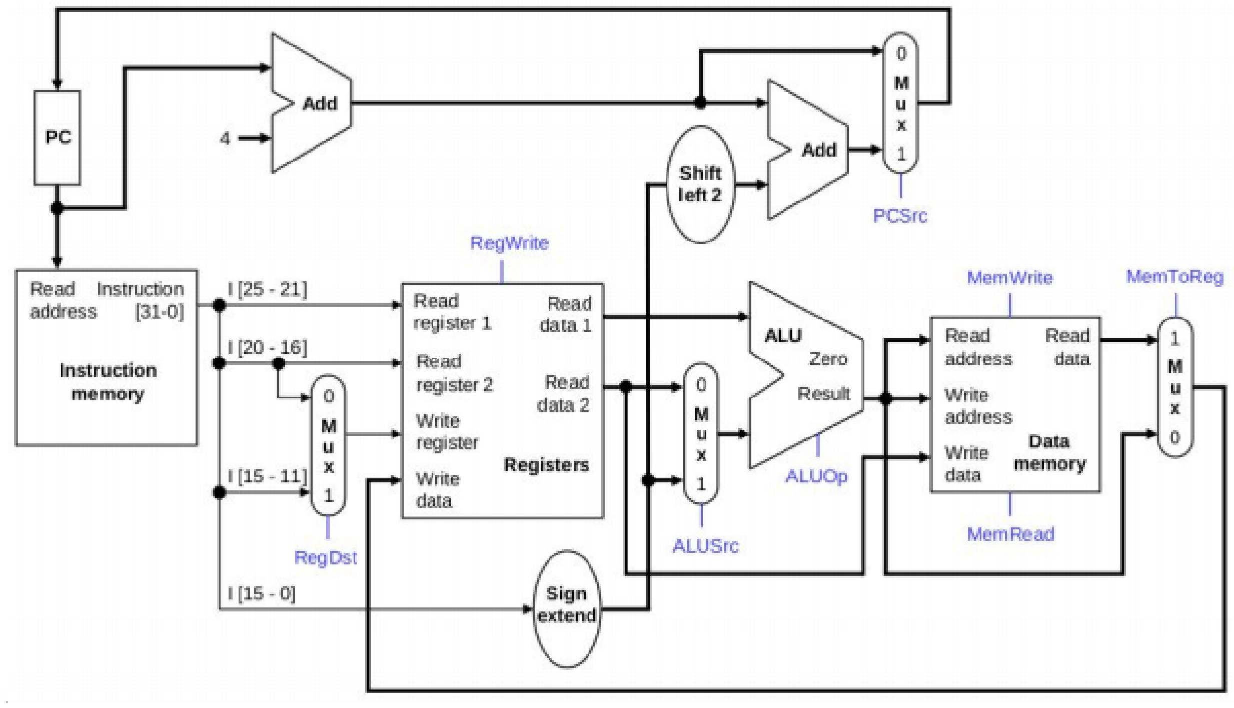
- An example load instruction is lw $t0, -4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



- Specific opns/ control codes set different elements as different instructions pass through

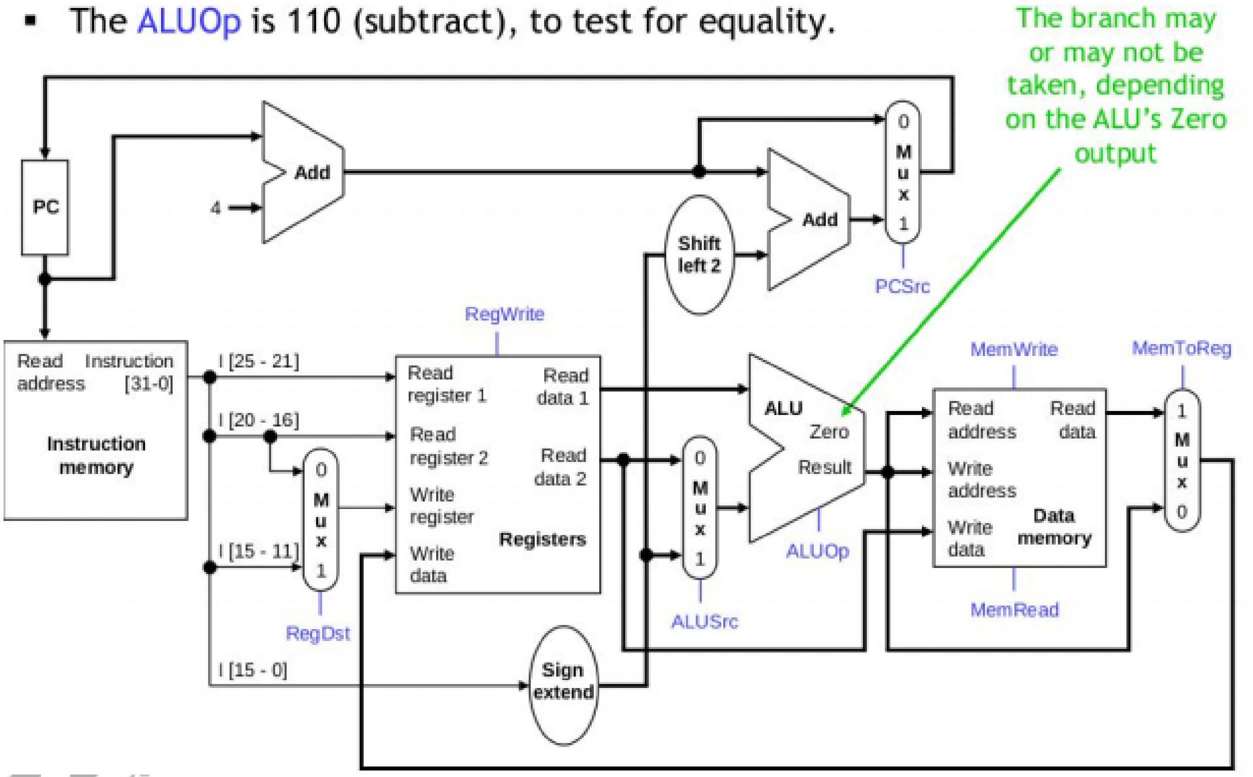
Recap: sw instruction path

- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.



Recap: beq instruction path

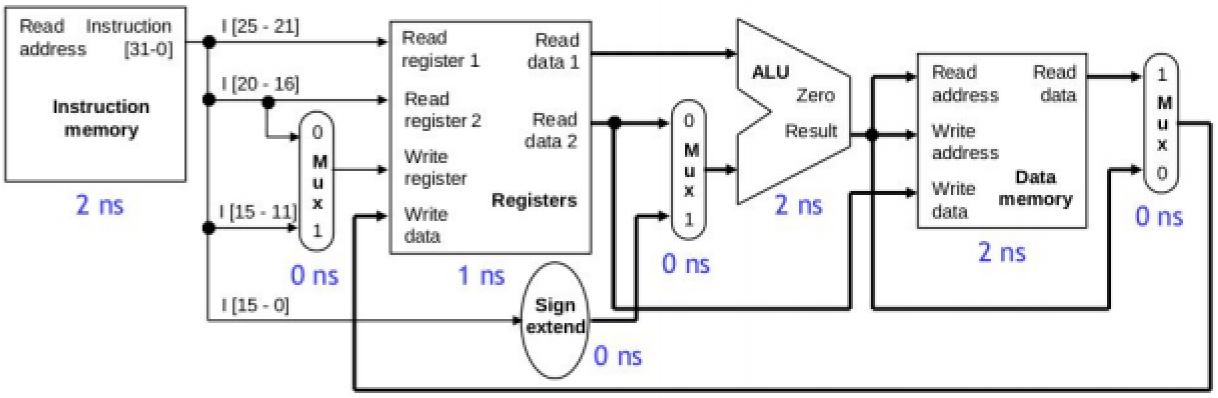
- One sample branch instruction is beq $at, $0, offset.
- The ALUOp is 110 (subtract), to test for equality.



Datapath and clock

List steps and calculate the total time to complete the following instructions through a single cycle datapath below: (i) lw $t0, 4($s0) (ii) add $s4, $t1, $t2

- E.g: **> reading the instruction memory - 2ns**
  > reading registers $x and $y - 1ns
  ...
  ... Total time - ...



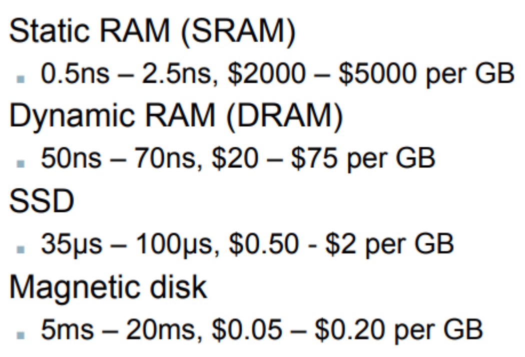
Memory

- 'The illusion of unlimited fast memory'
- What programmers really want, so we fake it
- Similar to going to the library
- Principle of temporal locality
    - Items accessed recently are likely to be accessed again soon
        - e.g., instructions in a loop, induction variables
- Principle of spatial locality
    - Items near those accessed recently are likely to be accessed soon
        - e.g., sequential instruction access, array data

Locality

- Taking Advantage of Locality
- Memory hierarchy – multiple levels of memory with different speeds and sizes
    - As distance from memory to processor increases, so does memory size and access time
        - In terms of library analogy: on desk, near you, there is little space and so have less books near you – further away on a bookshelf you may have many books (requires more time to get)
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
    - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
    - Cache memory attached to CPU

Memory Technology



Static RAM (SRAM)
- 0.5ns – 2.5ns, $2000 – $5000 per GB

Dynamic RAM (DRAM)
- 50ns – 70ns, $20 – $75 per GB

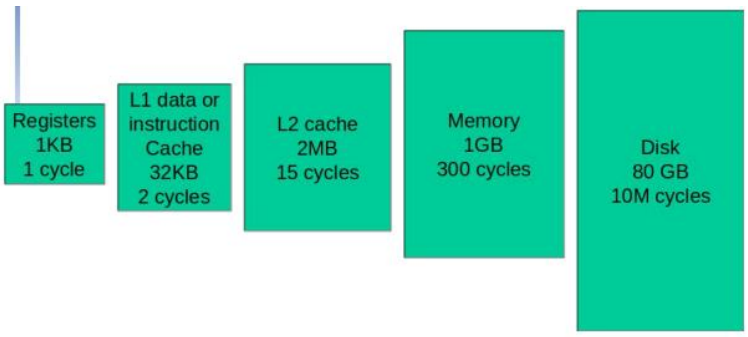
SSD
- 35µs – 100µs, $0.50 - $2 per GB

Magnetic disk
- 5ms – 20ms, $0.05 – $0.20 per GB

- Four primary technologies used in memory hierarchy:
    - Static RAM (SRAM): fastest (closest to processor), expensive
        - Volatile
    - Dynamic RAM (DRAM): faster, pricey
        - Volatile
    - SSD: slow, inexpensive
        - Used integrated circuits to store data (non-volatile)

- Magnetic disk: slowest, cheapest
  - Has moving parts
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Memory Hierarchy

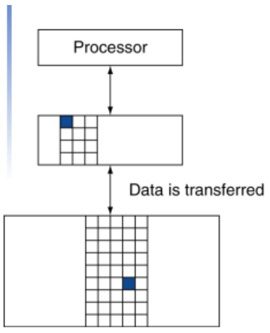


- As you go further, capacity and latency increase
- Register are on processor itself - contains variables that processor must compute/work on

CACHE HIERARCHIES

- Internet browsers also cache web pages – same concept
- Data and instructions stored on DRAM chips – DRAM has high bit density, which means can keep a lot of info in a small space
  - But relatively poor latency – need more time to access data from it
    - An access to data in memory can take as many as 300 cycles!
- Thus, some data is stored on the processor in the cache
  - Caches employ SRAM, which is faster, but has lower bit density
- All cache lvls integrated onto processor chip itself

MEMORY HIERARCHY LEVELS

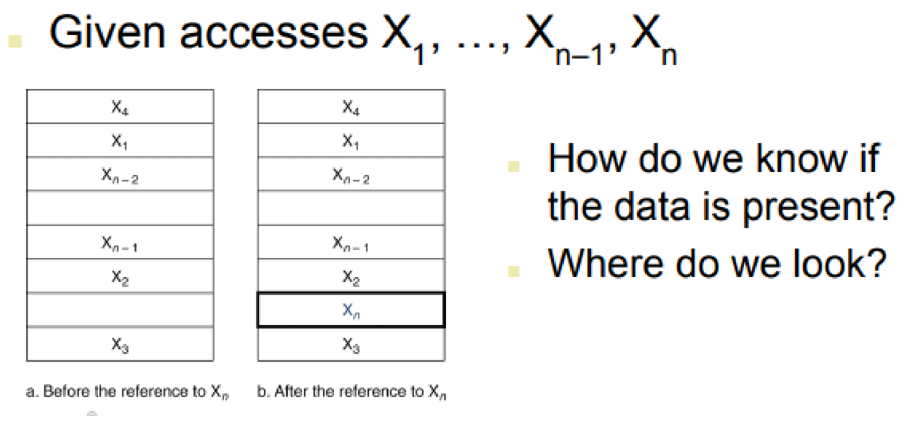


- Upper lvl memory => closer to processor (smaller and faster than lower level)
  - For every pair of levels in memory hierarchy, each has an upper and lower level
- Block (aka line): unit of copying (within each level)
  - May be multiple words
  - In terms of library analogy: go and get something from shelf, get book (a block) and take it back to desk

        - If need to get more books, then get more books (blocks)
    - Min unit of info that can be present/not present in cache
- If accessed data is present in upper level
    - Hit: access satisfied by upper level
        - Hit ratio: hits/ memory accesses
    - In terms of library analogy: sitting at desk and able to find info in one of the books in front of you
- If accessed data is absent
    - Miss: block copied from lower level (slower)
        - Time taken: miss penalty
        - Miss ratio: misses/memory accesses = 1 – hit ratio
    - Then accessed data supplied to requester from upper level
    - In terms of library analogy: have to get book from shelf *and* bring it to desk
- From performance perspective, time spent to service both hits and misses is important
    - Hit time: time required to access lvl of memory hierarchy, including time needed to determine whether access is a hit/miss
    - In terms of library analogy: time needed to determine whether book on desk is the one that you need and to get that book
    - Miss penalty: time required to fetch block from lower memory, including time to access block, transmit from one lvl to another, insert it into lvl that experienced the miss and pass block to requester.
    - In terms of library analogy: time needed to go to shelf, find the required book, bring it to the desk and present it to be accessed
- All factors – hit rate, hit time, miss rate, miss time – affect performance of processing.
- Memory systems critical to computer performance.
- Focus: how cache is structured and how info is mapped, found and moved bet hierarchies in order to minimize miss penalty/increase hit rate

Cache Memory

- The level of the memory hierarchy closest to the CPU


Given accesses $X_1, \ldots, X_{n-1}, X_n$

How do we know if the data is present?

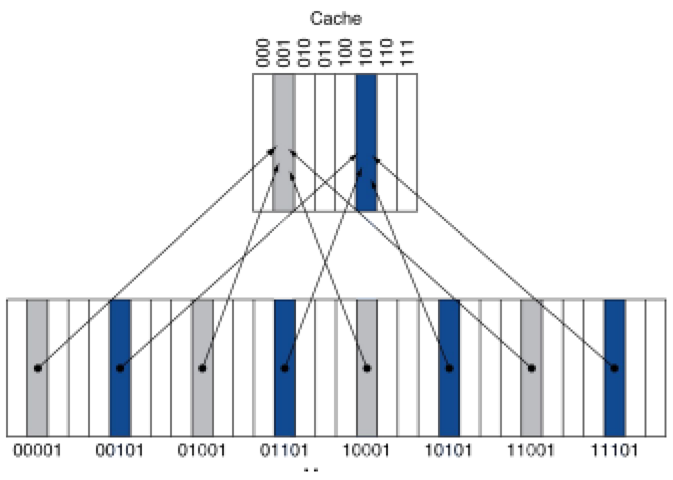
Where do we look?

- For example (diagram above)
    - (A) and (B) are arrays
    - (A) shows state before current request – cache contains collection of recently referenced items starting with X1, X2…to X(n-1)

        - Current request is for Xn
    - Processor requests Xn (word – i.e. data unit made up of 32 bits/4 bytes) – min unit we can request from memory hierarchies
    - Xn is not in cache – request results in miss
        - So Xn brought from memory into cache – shown by (B)
- Note: primary data is based on data in memory
    - Cache keeps small subset of data in main memory
    - Questions are then: how to know if data item from main memory location that has been referenced is actually available in the cache?
        - And if data is available in the cache, how to find it?
        - How to map from what has been referenced as mem address to a specific cache location to determine if that data is in the cache or not?

DIRECT MAPPED CACHE

- Simplest way to know if data item from mem location is in cache – assign each cache location based on address of word in mem
    - At word lvl, each mem location is a line to a specific entry in the cache
    - If have a mem reference, can check specific location in cache to see if it's there or not
- Location determined by address
- Direct mapped: only one choice
    - Cache structure is such that each mem location is mapped to exactly one location in the cache
    - (Block address) modulo (# of Blocks in cache i.e. cache size)



- # of Blocks is a power of 2
- Use low-order address bits

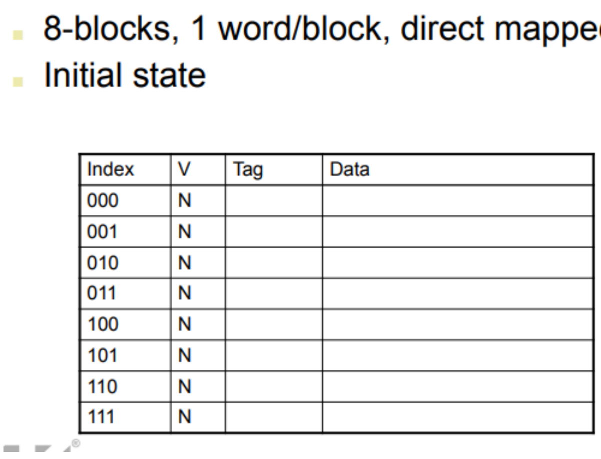
- In example, mem has 32 locations (0 – 31)
        - Each address has 5 bits (2^5 = 32)
    - Cache has 8 entries – each cache entry can be addressed by 3 bits (2^3 = 8)
    - There are 8 word entries in the cache – memory block maps to x % 8
        - x is block mem address
    - E.g for mem word location 9, cache entry is 9%8 = 1
        - Cache entry for address 9 is 1 (001 in binary)
- Several mem locations maps to same entry in cache (because of modulo)
- Index pointed to (i.e. cache entry) is lower order bits of mem address
    - In example, use lower 3 bits (since cache uses 3 bits for index) – log2(8) = 3

    - Lower order bits of mem index 9 is 001 (9 is 01101)
- Can thus find cache entry (in the example) either by:
    - Doing modulo opn or
    - By working out how many bits represent cache index and use lower order bits in given mem address – that will be mapping to that cache index

TAGS AND VALID BITS

- How do we know which particular block is stored in a cache location? – since several blocks map to same entry in cache
    - Store block address as well as the data – can look at cache entry and check if address stored there matches with what is being referenced
    - Actually, only need the high-order bits – since already have part of address in form of cache index
        - Called the tag bits
- What if there is no data in a location?
    - Valid bit: 1 = present, 0 = not present
    - Initially set to 0 (data entry invalid – shouldn't be used)

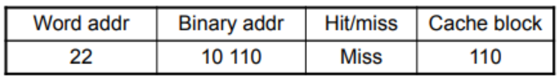
Cache Example



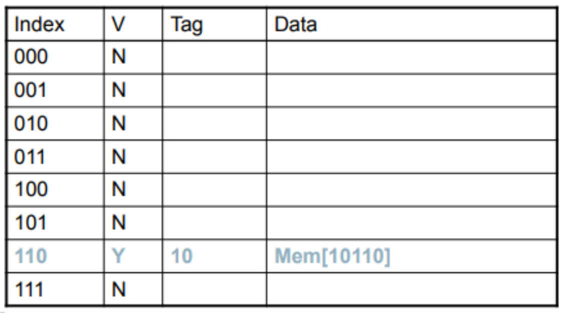
- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

- Each cache entry includes data and index #
- Each data entry made up of 1 word (32-bit data collection)
    - 3 bits for cache index
- In initial state of cache, valid bits set to 0
    - Tag fields will be empty

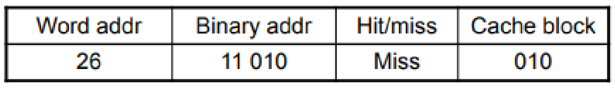


Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

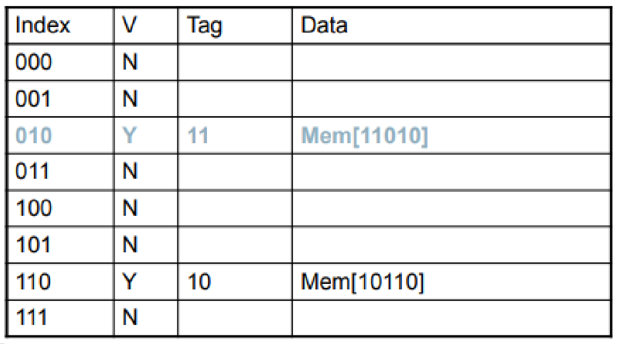


Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

- First access is for word 22 (address 10110)
    - Request is a **miss** since there is no data in cache entry 110
- Copy data from memory address into cache entry 110 and set valid bit to Y (i.e. has valid data)
    - 10 copied into field tag



Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010



Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

- Second access for word 26 (address 11010)
    - Again, a miss, followed by data copied

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

- Next request for words 22 and 26 result in a hit

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

- Then words 16 and 3 will result in misses initially
    - Handled the same as above
    - Another request for 16 results in a hit

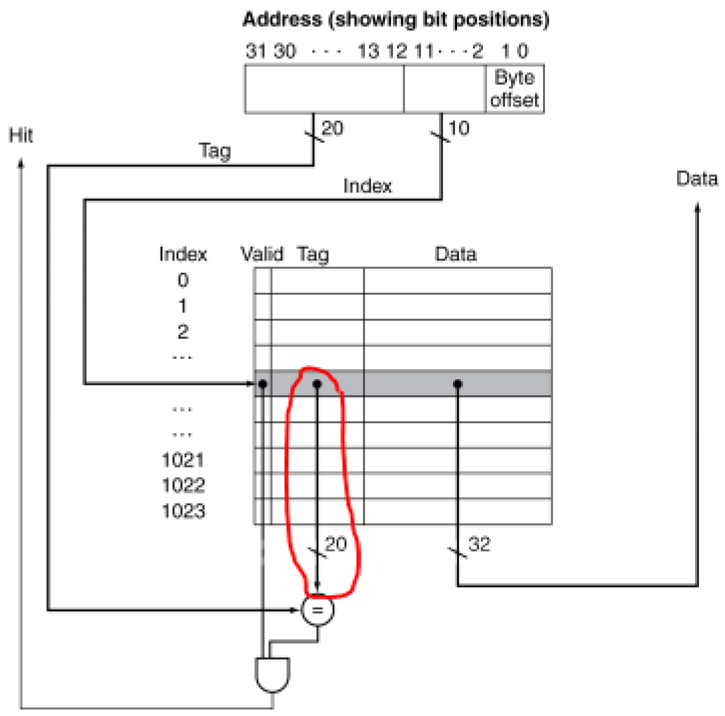
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

- Finally have word address for 18 (address 10010) – lower order bits point to entry that has *valid* data
    - But this is NOT a match – tags don't match (18 has upper bits 10 but entry has 11 as tag bits)
    - Thus, results in a **miss**

    - Entry for that address – data inside replaced by data from word memory address 18
    - 26 was there initially – now replaced, so subsequent requests for 26 results in a miss
- Matching based on index *and* tag

Address Subdivision



- Lower bits used to select cache entry index and entry consists of data and tag
- Example (diagram above):
    - Have indices 0 – 1023, which means size 1024
    - Log2(1024) = 10, means need 10 bits to represent index in cache
    - Addresses given per word (4-byte blocks) – address has lowest 2 bits as byte offset for pointing to specific byte in that word
        - Address will be shifted left 2 in order to point to words – lower order bits could be used to point to specific byte location
    - 32-10-2 = 20 bits, used for the tag
    - Tag from cache used to compare upper 20 bits of address to determine whether entry in cache corresponds to specific address (red outline)
        - If tag matches upper 20 bits are equal, check that valid bit is set to 1 – request is then a hit

Larger Block Size

- What if each block holds multiple words?
    - Want to map mem address to multi-word cache block
- Example:
    - Cache with 64 blocks, 16 bytes/block
        - Block size of 4 words
    - To what block number does address 1200 map?
        - Block address = $\lfloor 1200/16 \rfloor$ = 75 (byte address/bytes per block)
        - Block number (index) = 75 modulo 64 = 11

        - Log2(64) = 6, use 6 bits for index
        - 16 bytes in each block, log2(16) = 4, use 4 bits for byte offset

BLOCK SIZE CONSIDERATIONS

- Why should a single block include more than one word?
- Larger blocks should reduce miss rate
    - Due to spatial locality
    - {Note to self – ignore this until go through textbook cause video (4c) makes no sense [04:23 timestamp] NB that larger blocks *exploit* spatial locality – each time copy from memory to cache, take more bytes (for example, it would be more than just 4 bytes)
        - By doing this and increasing block size}
- But in a fixed-sized cache
    - Larger blocks ⇒ fewer of them
        - More competition for entry into blocks, leads to higher block replacement rate ⇒ increased miss rate
    - Larger blocks ⇒ pollution
        - More unnecessary data is loaded into cache
- Larger miss penalty
    - Can override benefit of reduced miss rate
    - Early restart - allow resumption of CPU execution as soon as requested word or block is returned
        - Instead of waiting for entire block to be copied into cache, resume execution when have needed word in the processor.
    - …..and critical-word-first can help
        - Critical word first: get the requested word first from the memory, send it asap to the processor and then continue reading the rest of the block into the cache

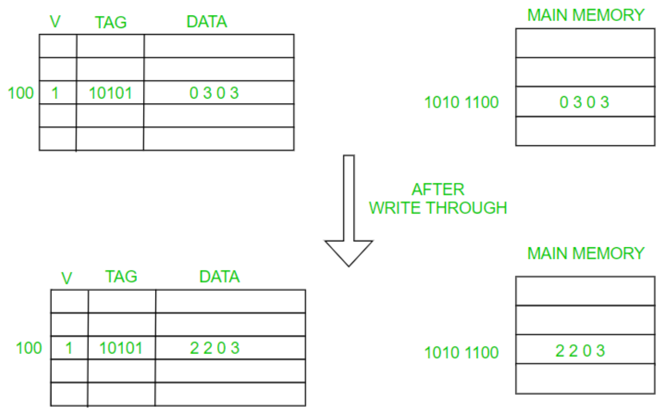
CACHE MISSES

- Cache miss: request for data from cache not fulfilled because data may not yet be present in the cache
- On cache hit, CPU proceeds normally
- On cache miss
    - Stall the CPU pipeline
        - Stalling => freezing state of contents of regs at that point, while wait for data to be fetched from lower lvl memory
    - Fetch block from next level of hierarchy
    - Instruction cache miss
        - Restart instruction fetch
    - Data cache miss
        - Complete data access

WRITE-THROUGH

- Mem writes work differently from mem fetch
- Whenever a Processor wants to write a word, it checks to see if the address it wants to write the data to, is present in the cache or not. If address is present in the cache i.e., Write Hit.

- On data-write hit, could just update the block in cache
    - But then cache and memory would be inconsistent
    - To avoid this, always write data into *both* memory AND cache
        - This scheme is write through
- Write through: also update memory
    - But makes writes take longer – could overall end up significantly slowing down processor
        - e.g., if base CPI (without cache miss) = 1, 10% of instructions are stores, i.e. write to memory, takes 100 cycles
            - Effective CPI = 1 + 0.1×100 = 11
    - Solution: write buffer
        - Holds data waiting to be written to memory
        - CPU continues immediately
            - Only stalls on write if write buffer is already full
            - As soon as get a miss and thus write instruction, once it's written to cache, continues to proceed with execution while buffer data continues to be written to memory
                - Implications i.t.o previous example – don't have to wait 100 clock cycles to complete before processor can continue with whatever it was doing
                - Let buffer deal with finalization of writing data to memory – processor can continue



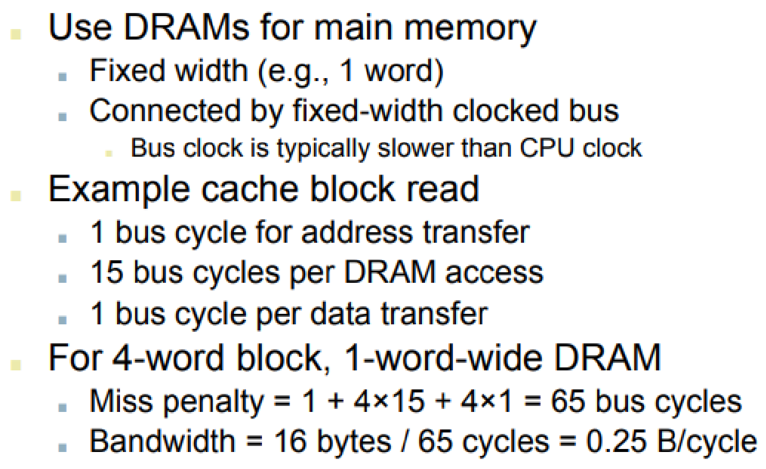
WRITE-BACK

- Alternative approach: On data-write hit, just update the block in cache
    - Don't write new value to lower mem
    - Keep track of whether each block is "dirty" i.e. track cache blocks that have been modified
        - Since blocks are now inconsistent with lower level
- When a dirty block is replaced in the cache
    - Write it back to memory
        - Don't have to write back to memory all the time – only if content written to cache needs to be replaced
    - Can use a write buffer to allow replacing block to be read first
        - i.e. new block read first before "dirty" blocks written down to memory

WRITE ALLOCATION

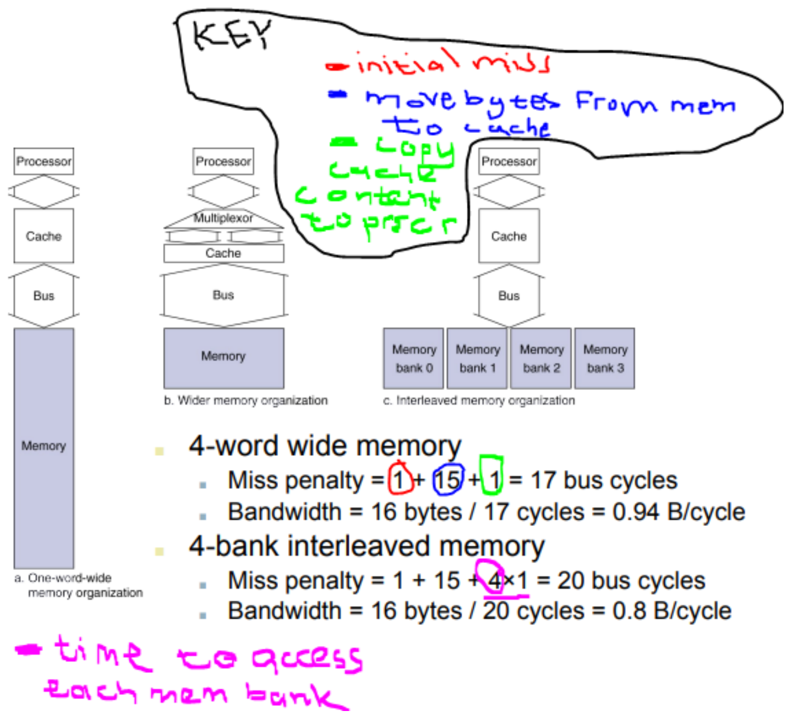
- What should happen on a write miss?
    - Write Miss: Write occurs to a location that is not present in the Cache
- Alternatives for write-through
    - Allocate on miss: fetch the block
        - In Write Allocation data is loaded from the memory into cache and then updated.
    - Write around: don't fetch the block
        - Since programs often write a whole block before reading it (e.g., initialization)
        - Data is directly written/updated to main memory without disturbing cache.
            - It is better to use this when the data is not immediately used again.
- For write-back
    - Usually fetch the block

MAIN MEMORY SUPPORTING CACHES



- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

Increasing Memory Bandwidth



a. One-word-wide memory organization

b. Wider memory organization

c. Interleaved memory organization

KEY

- initial miss
- move bytes from mem to cache
- copy cache content to proc r

- 4-word wide memory
  - Miss penalty = $\boxed{1} + \boxed{15} + \boxed{1}$ = 17 bus cycles
  - Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle
- 4-bank interleaved memory
  - Miss penalty = 1 + 15 + $\underline{\boxed{4}\times 1}$ = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle

- time to access each mem bank

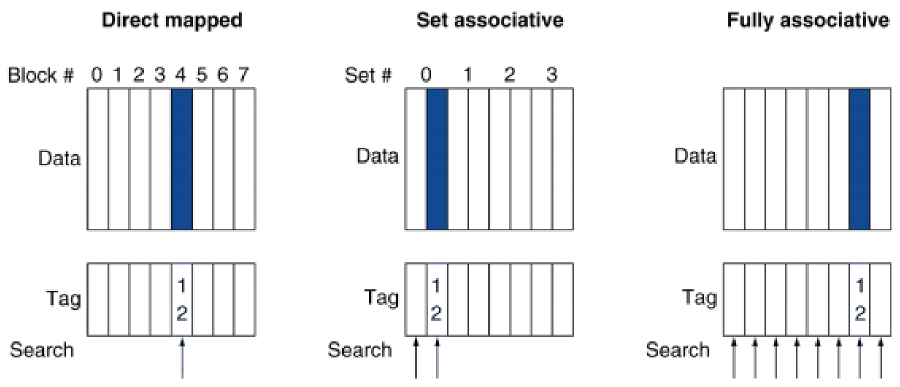
- Another approach to improving performance while writing to memory – increase bandwidth of memory buses
  - i.e. minimize performance loss due to memory access by increasing bandwidth of main memory in order to transfer cache blocks more efficiently.
- Can:
  - Make memory bus wider
  - Use interleaving
- Want to reduce # of cycles needed to move bytes from memory to cache
  - Interleaving achieved by spreading mem addresses evenly across mem banks – such that, successive words in address space are placed in different modules
  - Consequently, successive mem reads and writes will use different memory banks in turn

        - Reduces time spent waiting for memory banks
    - Overall results in higher mem throughput i.e. amount of data sent per cycle
- Example:
    - 4-word wide memory – use 17 cycles to get x# bytes into processor
    - 4-bank interleaved memory – use 20 cycles to get x# bytes into processor
- Different approaches yield different throughputs and achieves different performances

Associative Caches

- Fully associative
    - Allow a given block to go in any cache entry
    - Requires all entries to be searched at once in order to find a given block
        - Search done in ||
    - Comparator per entry (expensive)
- n-way set associative
    - Fixed # of locations where memory block can be mapped to
        - Compromise between direct mapping (each block can only be mapped to specific location in cache) and fully associate mapping (block can be mapped to any location in cache)
    - Each set contains n entries
        - Each block in mem maps to unique set in the cache – given by index field
    - Block number determines which ***set*** (key thing to note – maps NOT to a unique field but rather a set of blocks)
        - (Block number) modulo (# of sets in cache)
        - Block can be placed in any element of set that it maps to
    - Search all entries in a given set at once
    - n comparators (less expensive than comparator for each entry)

Example

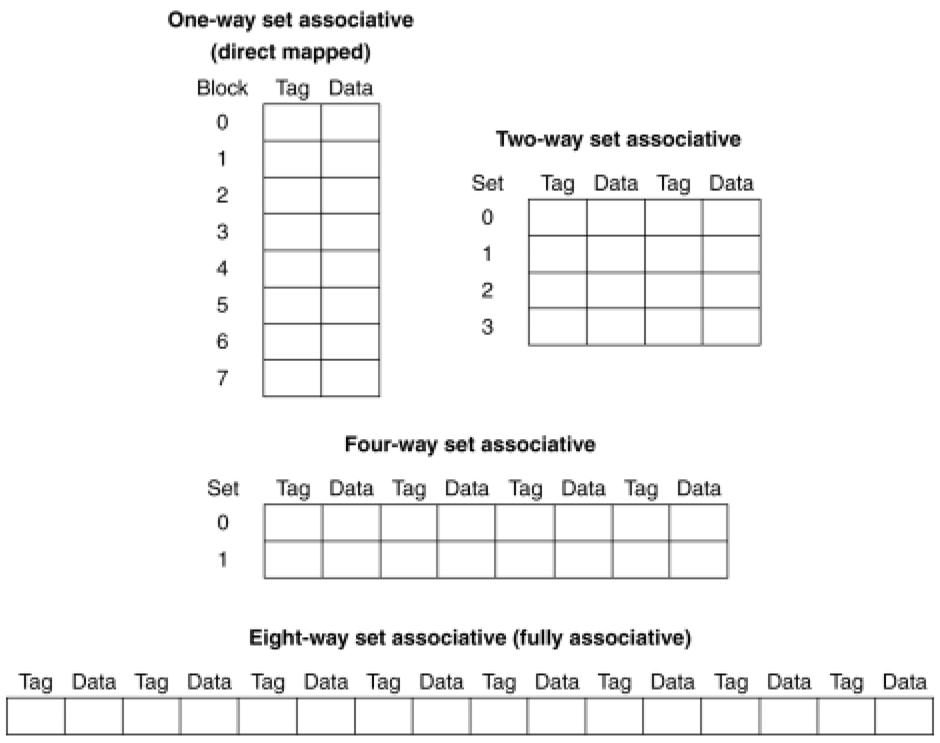


- Mem block with address 12 situated in cache with 8 blocks
    - Arrows indicate where mapping to
- Direct mapped:
    - Only *one* cache block where mem block 12 can be found
    - 12 % 8 – since 8 entries in the cache, used to locate mem block 12
        - Block 12 thus is in entry 4 of the cache
- Set associative (two way):

- Have 4 sets - each set with two entries
    - 8 (total # entries) /2 (# entries in each set) = 4, thus can have 4 sets
    - 12 % 4 = 0, means mem block 12 should be in set zero
        - Can be in any element in set
    - Check set until find block 12
- Fully associative:
    - Mem block 12 can be placed in any of 8 cache entries
    - Arrows represent comparator locations at each point
    - Check all entries at the same time
- Use tag to check entries (for all three above)

SPECTRUM OF ASSOCIATIVITY

- Example - for a cache with 8 entries, can see different configurations for different levels of associativity


One-way set associative
(direct mapped)
Block Tag Data
0 1 2 3 4 5 6 7
Two-way set associative
Set Tag Data Tag Data
0 1 2 3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0 1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

- Total cache size in block = # sets x associativity
    - Increasing associativity => # elements per set increases, BUT total # sets decreases
    - Direct map – one-way associative cache
        - Max # of sets (8) – each set is one entry
    - Set associative – two way
        - Has 4 sets
        - Four way has 2 sets
    - Fully associative – eight way
        - Has 1 set

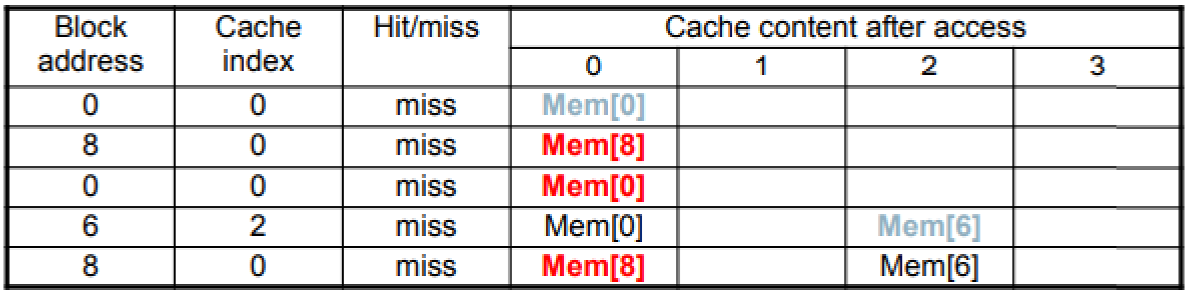
Example

- Which lvl of associativity results in better performance/ better hit rate?
- Compare 4-block caches

- Direct mapped, 2-way set associative, fully associative
- Block access sequence: 0, 8, 0, 6, 8
- Compare using # of misses

Direct mapped

- Cache mapping – block address % # of entries i.e. 4 (4 entries in cache)
  - 0 % 4 = 0
  - 6 % 4 = 2
  - 8% 4 =0
- Looking for entries 0,2



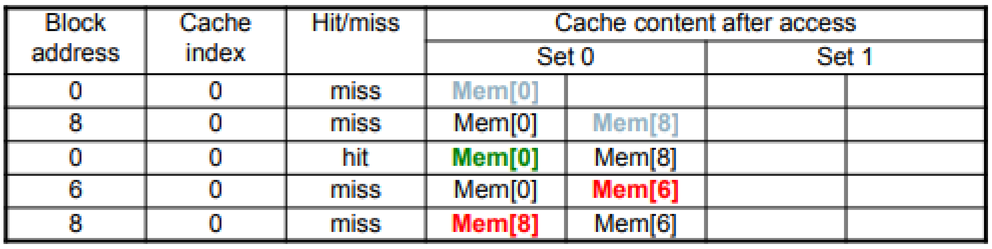
Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	**Mem[8]**		Mem[6]	

- Procedure:
  - Fill in cache contents after each reference – e.g. after zero first requested
  - Check cache entry – copy contents from main memory into cache
  - Blank entry = invalid block
- Red font => new entry added to cache for that reference
  - Plain text => old entry, hasn't been updated since last access
- Process:
  - Check entry 0 – it is a miss, since cache entry is empty
    - Copy contents of memory block zero copied into cache index 0
  - Check entry 0 again – it is a ***miss***, since we're looking for address 8 but address zero is stored there
    - Copy contents of memory block 8 copied into cache index 0
  - Check entry 0 – it is a miss, since cache contents are for block 8
    - Copy contents of memory block zero copied into cache index 0
  - Check entry 2 – it is a miss, since cache entry is empty (first time referencing block 6)
    - Copy contents of memory block six copied into cache index 2
  - Check entry 0 again – it is a ***miss***, since we're looking for address 8 but address zero is stored there
    - Copy contents of memory block 8 copied into cache index 0
- Take away – direct mapping results in "aggressive" replacements
  - Multiple memory blocks point to same location in cache – in many sequences can end up with replacements because many addresses point to same cache entry
  - More requests = more replacements = more misses

Set associative (2-way)

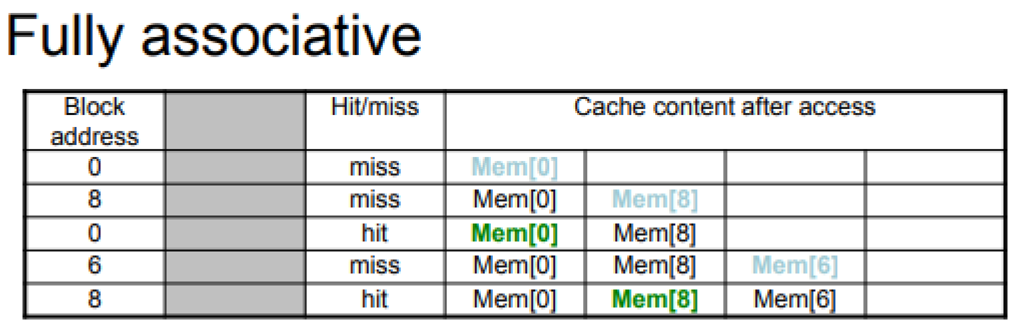
- Have 2 sets – 4 entries / 2 blocks per entry
- Each set has first index 0, second index 1
  - i.e. 2 elements can be put in the set

- Determine set for given block – block # % # of sets
    - 0 % 2 = 0
    - 6 % 2 = 0
    - 8 $2 = 0
        - All map to same set
- Multiple entries per set gives choice as to which entry in a set to replace
    - Replace the <u>least recently</u> used block within a set
- Process:
    - Check set 0 – it is a miss for 0, since set is empty
        - Copy contents of memory block zero copied into set 0, index 0
    - Check set 0 again – it is a ***miss***, since set 0 does not contain 8
        - Copy contents of memory block 8 copied into set 0, index 1
    - Check set 0 – it is a <u>hit</u>, since set 0 index 0 contains zero
    - Check set 0 – it is a miss, since set 0 does not contain block 6
        - Copy contents of memory block 6 copied into set 0, index 1 – replacing block 8 since it is the least recently used block within the set
    - Check entry 0 again – it is a ***miss***, since we're looking for address 8 but address zero is stored there
        - Copy contents of memory block 8 copied into set 0, index 0 – for same reason as why we replaced 8
- 1 hit, 4 misses
    - Slight improvement



Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Fully associative



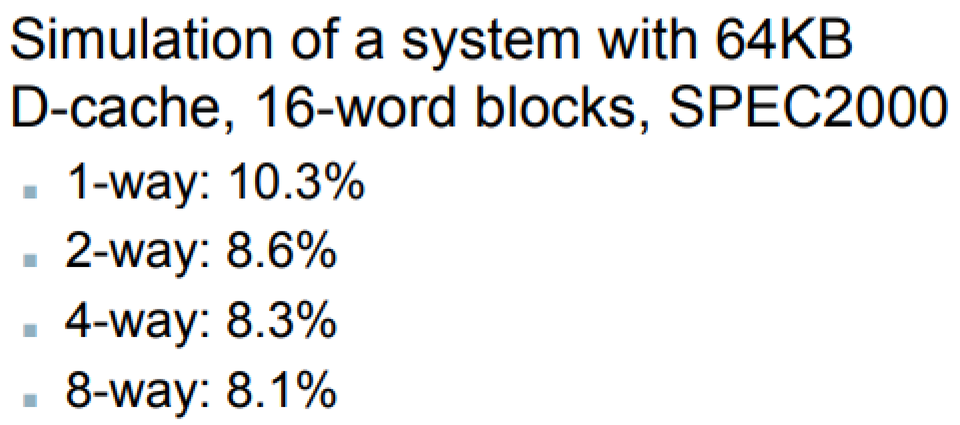
Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

- Since memory blocks can be placed in any of the cache entries, they are just copied in linearly
- 2 hits, 3 misses (best…so far)

HOW MUCH ASSOCIATIVITY?

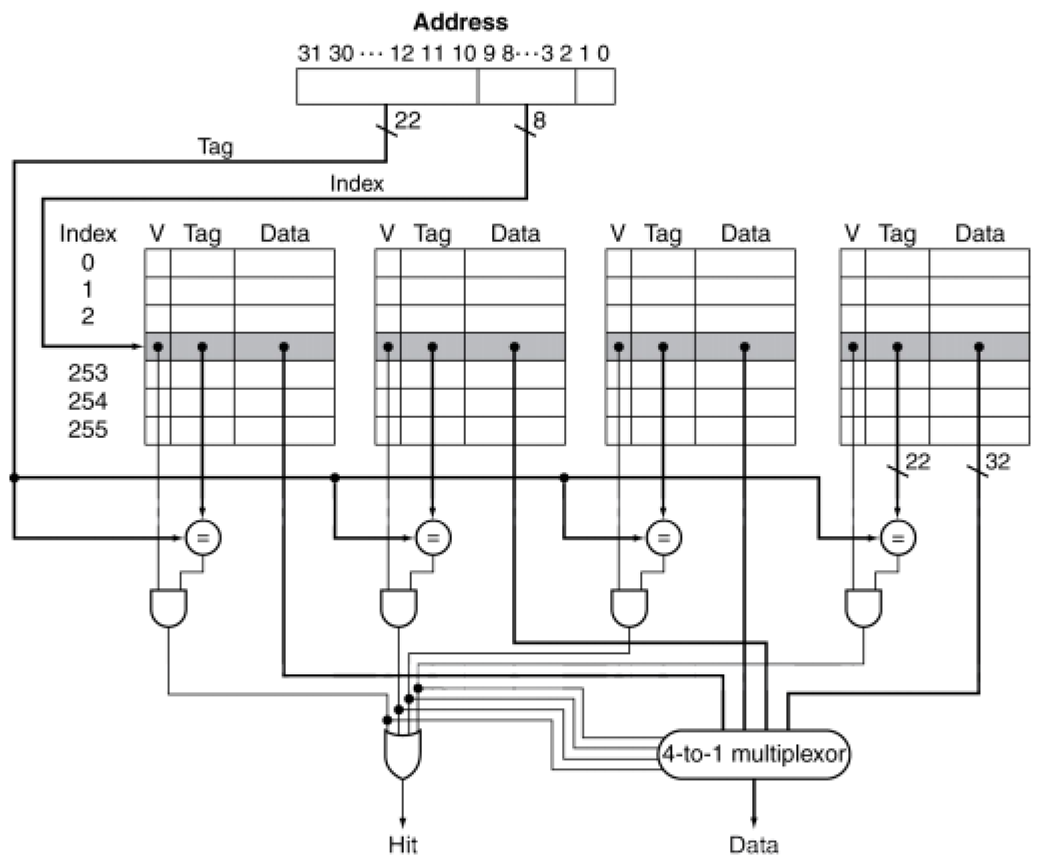
- Increased associativity decreases miss rate
  - But with diminishing returns
- Proof (done experimentally)
  - Following table shows % decrease in miss rate for each added "way" of associativity
  - Can see diminishing returns – between 4-way and 8-way especially



Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000

- 1-way: 10.3%
- 2-way: 8.6%
- 4-way: 8.3%
- 8-way: 8.1%

SET ASSOCIATIVE CACHE ORGANIZATION



- Finding a block in a set associative cache:
  - Index value used to select set containing address of interest – above diagram has 256 entries (0 – 255)

    - To represent 256 index values => need 8 bits (2^8 = 356)
    - Two lowest order bits are used as block offset
        - Index bits thus 8 bits *after* two lowest order bits
- Each block includes tag – indicates block address when combined with index and block offset
    - 22 remaining bits (32 – 2 – 8 = 22) are the tag for diagram above
- In 4-way set associative cache, 4 comparators needed
    - And 4-to-1 multiplexor – helps to choose matching tag amongst potential members of the set
- Once index entry found, examine tag of every block within set to see if it matches with upper bits of block address requested by processor
    - Comparator applied to each tag
- If valid bit true and tag matches = hit
    - Send positive data selection signal to data multiplexor
- Data from matching element allowed through multiplexor and sent through to processor/higher level in the memory structure
- Cost of associative cache includes comparator that must be applied on each set
    - Additional delays for comparing tags bits in each set too

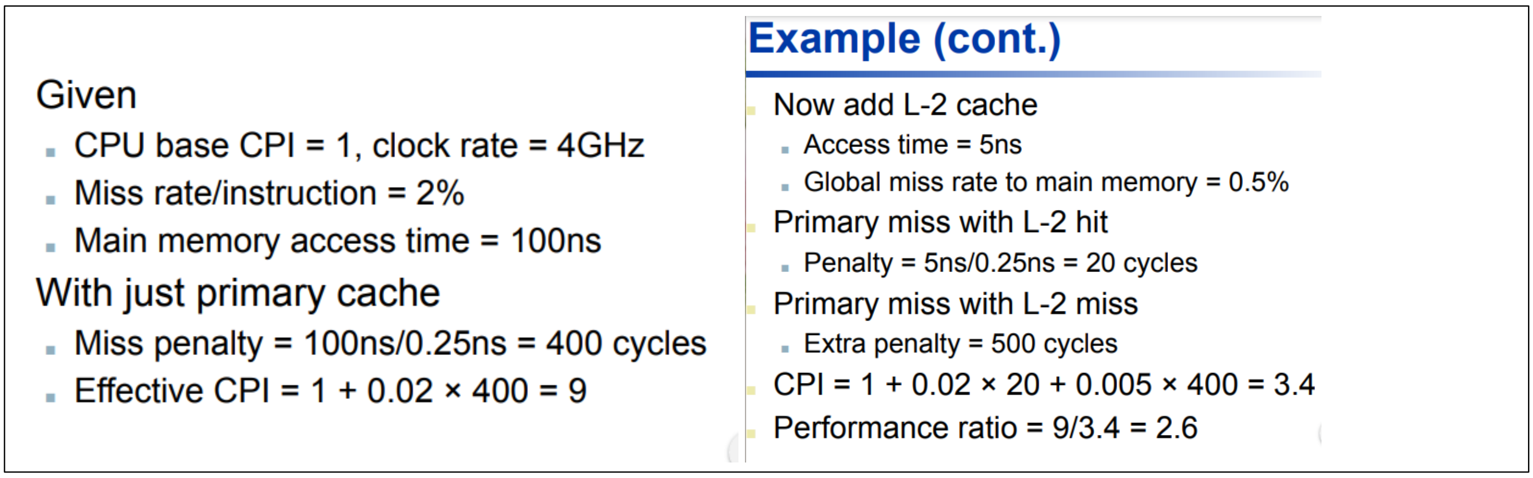
REPLACEMENT POLICY

- Direct mapped: no choice
- Set associative
    - Prefer non-valid entry, if there is one
    - Otherwise, choose among entries in the set
- Least-recently used (LRU)
    - Choose the one unused for the longest time
        - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
    - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

- Primary cache attached to CPU
    - Small, but fast
- Level-2 cache services misses from primary cache (placed on same chip as processor)
    - Miss penalty for L1 cache is the time to access L2 cache
        - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Example



Given

- CPU base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns

With just primary cache

- Miss penalty = 100ns/0.25ns = 400 cycles
- Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

- CPI = 1 (on avg 1 clock cycle for each instruction IF all references are a hit in primary cache)
- Main memory access time = 100 ns (includes time to handle a miss)
- Miss rate/instruction at primary cache = 2% => for all instructions processed, 2% will miss
- Miss penalty i.t.o. CPU time is 100 ns but need it i.t.o clock cycles – CPU time is # clock cycles x cycle duration
  - Duration of each cycle is reciprocal of clock rate – in the example this = 4GHz
    - 1/4Ghz = 0.25 ns
  - Thus, working backwards, we say 100/0.25 = 400 clock cycles
- Since only have to access main memory 2% of the time, can work out CPI
  - To find effective CPI, 1+ 2% x 400 i.e. 1 + 0.02% * 400 = 9
- How much faster with added secondary cache?
  - Assume access time 5ns for hit/miss
- L2 cache larger enough that global miss rate reduced to 0.5%
- Miss penalty i.t.o clocks now 5/0.25 = 20 cycles for L2 hit
- L2 miss (miss needs to go to main memory)
  - Extra penalty – L2 cache access time + main memory = 400 cycles
- Since change to global miss rate, can again work out effective CPI: 1 + 0.02 *20 (L2 cache) + 0.005 * 400 (main memory) = 3.4
  - Much smaller CPI secondary cache
- Performance ratio = 9/3.4 = 2.6
  - Processor with secondary cache is 2.6 x faster

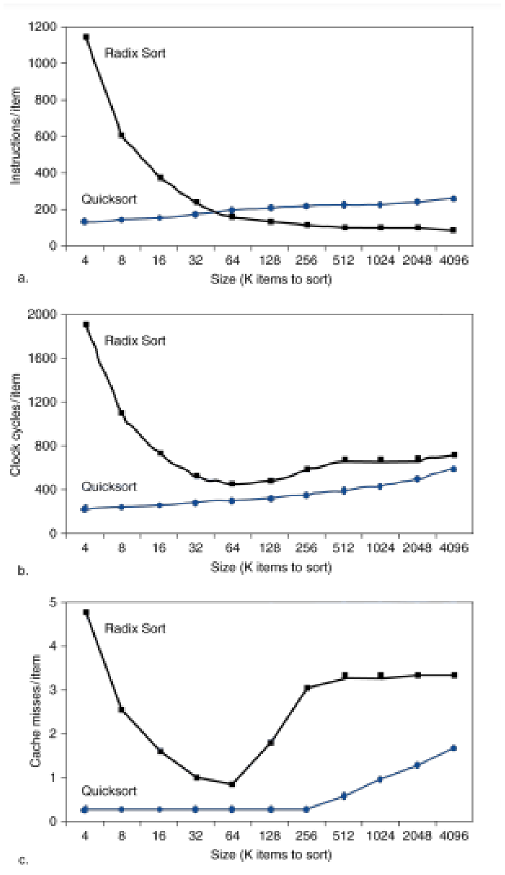
MULTILEVEL CACHE CONSIDERATIONS

- Multilvl cache is mem hierarchy with multiple lvls of caches
- Design specifications must consider all lvls of caches
- Primary cache
  - Focus on minimal hit time and fast access time
- L-2 cache
  - Focus on low miss rate to avoid main memory access (slower)
    - Design L2 cache using larger clock sizes – higher spatial locality
      - Getting more things from main memory and placing in cache – reduces miss rate
  - Hit time has less overall impact

    - Increase associativity – also reduces miss rate (less aggressive replacements)
- Results
    - L-1 cache usually smaller than a single cache
    - L-1 block size smaller than L-2 block size

INTERACTIONS WITH SOFTWARE

- Cache performance depends on processor interactions with the software
- Misses depend on memory access patterns, which depend on:
    - Algorithm behavior
    - Compiler optimization for memory access
- Cache optimization (basic idea): use all data in cache block repeatedly before replacing on a miss
    - i.e. Something in cache should be used as much as possible for as long as possible before replacement
        - => satisfying all data needs in cache (improves performance)
- Example graphs:



    - # of instructions executed by item searched, for different algos
    - Different # of instructions executed for each item
    - Quicksort spends fewer clocks per item that has to be sorted and shows fewer misses per item
- Graphs show way algo works and how it accesses different elements affects overall performance
- Want compiler and algo (and have app be structed) in such a way that most of the time you use content already in the cache

    - Alternatively, can think of it like “if you put something into cache, want to use it for as many instructions as possible before replacing it with something from main memory.”