Ex. hardware abstraction

for(...){

doWork(data[i]);

}

Mem request data[0] → response “0x1” → CPU execute doWord(“0x1”)...

CPU instruction 0.3ns

DRAM access 10-100 ns

→ CPU is always idle → bad architecture

CPU sockets and DIMMS per sockets

8 Great Ideas

1. Design for Moore’s Law
2. Use abstraction to simplify design
3. Make the common case fast
4. Performance via parallelism
5. Performance via pipelining
6. Performance via prediction
7. Hierarchy of memories
8. Dependability via redundancy

# IDEA #2 - Use abstraction to simplify design

Program Flow

application software(high-level)

system software (compiler, OS)

hardware (processor, I/O)

ISA (Instruction-Set Architecture) is the abstraction between software and processor hardware

consistent ISA allows app to run on diff machines of the same architecture (eg. x86 across Intel)

ISA is the agreement on what Binary machine language program should do (RISC)

Ex. 00A9 8933 = add values in registers x19 x10, and store it in x18

high-performance of ISA (usually measured by execution time)

elapsed time = total response time

CPU time (time spent by processor)

= clock cycle \* clock period = CPU clock cycles / clock rate

⇒ to improve performance, we can reduce # clock cycles or instead clock rate

Instruction Count for a program

Determined by program, ISA and compiler

Average cycles per instruction

Determined by CPU hardware

If different instructions have different CPI

Average CPI affected by instruction mix

Clock Cycles = Instruction Count \* Cycles per Instruction

CPU Time = Instruction Count \* CPI \* Clock Cycle Time

= Instruction Count \* CPI / Clock Rate

CPU Time Instruction Count CPI Clock Cycle Time

Clock Cycles Instruction Count Cycles per Instruction

* CPU clock = Operation of digital hardware governed by a constant-rate clock
* Clock period: duration of a clock cycle o e.g., 250ps = 0.25ns = 250×10–12s
* Clock frequency (rate): cycles per second o e.g., 4.0GHz = 4000MHz = 4.0×109Hz

Summary

* ⇒
* Low instruction count ⇒ Each instruction should do more work
* Low CPI High clock speed ⇒ Each instruction should be simpler

# IDEA #3 - Make the common case fast

Reduced Instruction-Set Computer (RISC) usually better option

Small number of more general instructions. Complex operations implemented by composing general ones

Simpler design allows faster clock

Simpler design allows efficient microarchitectural techniques

(eg. superscalar, out-of-order…)

Compilers very good at optimizing software

Complex Instruction-Set Computer (CISC)

Precise definition is debated (Not RISC?)

Many, complex instructions

Most modern CISC processors have RISC internals

CISC instructions translated on-the-fly to RISC by the front-end hardware

Added overhead from translation (silicon, power, performance, …)

RISC-V

Composable, modular design

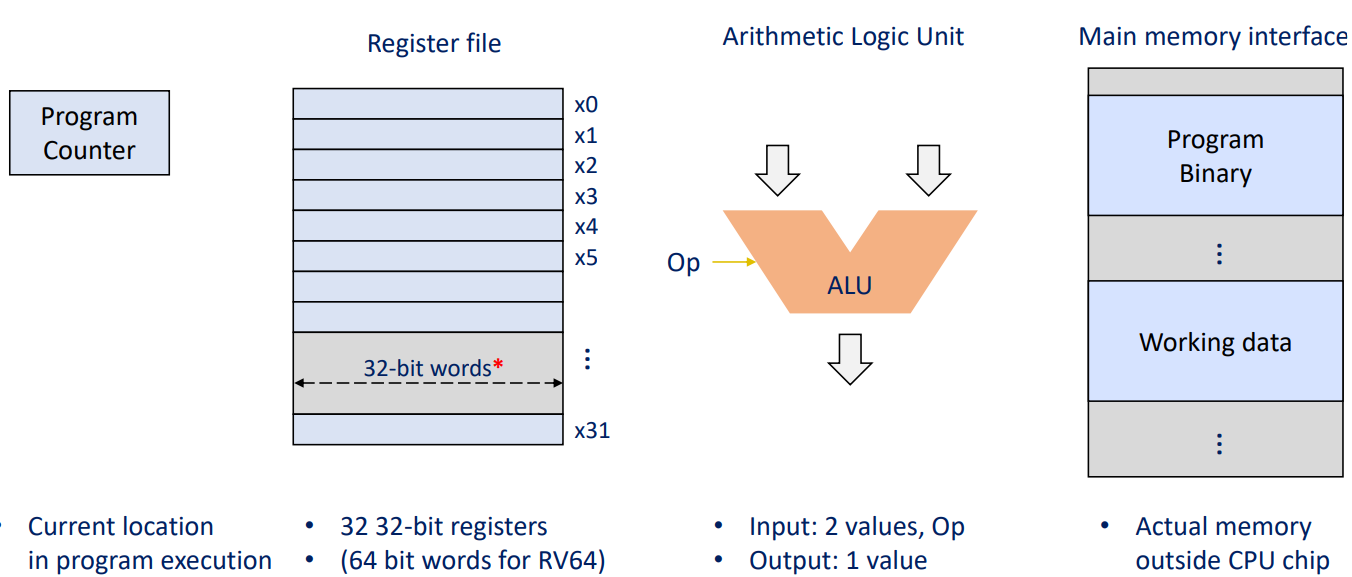
fixed-size registers (32-bit or 64-bit) **WHY not larger?**

3 types of instructions

computational operations: + – \* / in ALU

load/store

control flow: if/else, loop



processor job:

inst = mem[PC] //access memory

next\_PC = PC + 4 //access the next

//load/store/ALU/control flow

if ( inst.type == STORE ) mem[rf[inst.arg1]] = rf[inst.arg2]

if ( inst.type == LOAD ) rf[inst.arg1] = mem[rf[inst.arg2]]

if ( inst.type == ALU ) rf[inst.arg1] = alu(inst.op, rf[inst.arg2], rf[inst.arg3])

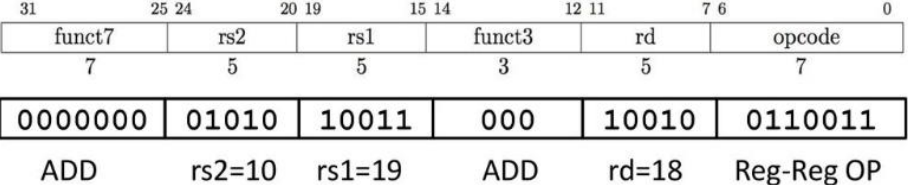
if ( inst.type == COND ) next\_PC = rf[inst.arg1]

//go to the next PC

PC = next\_PC

type, arg1, arg2, arg3, op

SA design determines how many bits represent each

Ex. 

Intel x86 - registers

Four ‘general purpose’ registers

Naming has historical reasons

Originally AX…DX, but ‘Extended’ to 32 bits (sometimes, some bit can be used as general purpose reg)

64 bit extensions with ‘R’ prefix

Special registers for stack management

RISC-V has no special register (Except x0)

addressing mode

Source/dest operand Second source operand

Register Register

Register Immediate

Register Memory

Memory Register

Memory Immediate

Ex. add <reg>, <mem>

Ex. add $10, %eax — EAX is set to EAX + 10

Ex. addb $10, (%eax) — add 10 to the single byte stored at memory address stored in EAX

backward compatibility + add more instructions ⇒ instruction accumulation

Instruction in ISA

RISC (simpler, fewer, general) — CISC, x86 (complex, more, specialized)

RISC-V encoding

4 bytes per instruction

diff instructions have diff param (reg, immediate, …)

various fields should be encoded to consistent location

ANSWER ?? (Simplicity, ease of decoding, reduce complexity of hardware)

x86 encoding

variable-length encoding, 1 to 1 bytes encoding

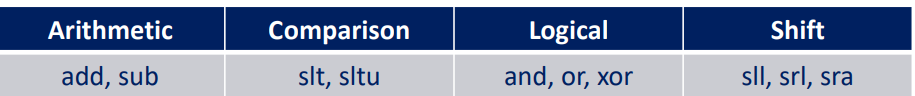
complex decoding logic to translate instruction to simpler micro op

RISC-V

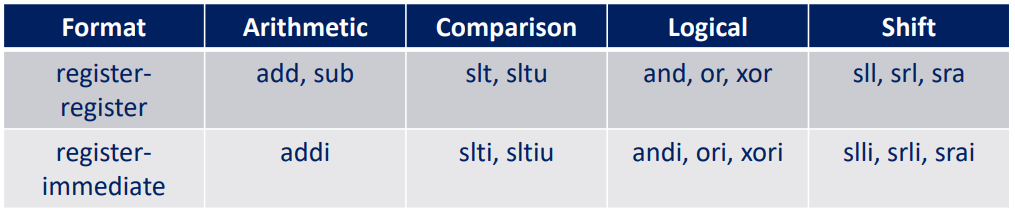
why not shift left? because shift right is the same logic as the shift left.

Encoding

R-Type (reg-to-reg?)



I-Type (reg-imm)



S-Type (store) two reg input, no output

sw src, offset(base) #store word

UJ-Type (jump link) one dest reg, one imm operand

some instructions need “immediate” values (ex.addi x1, x2, 32 – an immediate value)

12-bit register

in RISC? simpler encoding ⇒ U-type: LUI (load upper immediate) to fit into 32-bit spot??

in CISC: complex, variable-width data to encode ⇒ immediate value can use 0-4 bytes to encode

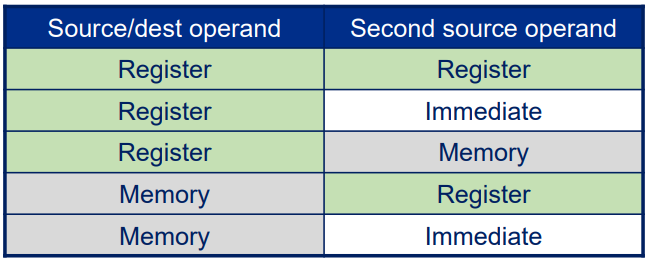
RISC

simplicity, regularity, and efficiency in instruction execution ⇒ fixed instruction formats + limited addressing modes + straightforward immediate value handling.

CISC

complex + variable-length instructions ⇒ powerful instructions + minimize # instructions for a task + slow down the clock

arithmetic + load/store



Transistor

with V ⇒ connect. Without V ⇒ break the circuit

0 and 1 are interpreted from V with threshold

noise (eg. V\_out is barely lower than V\_l or higher than V\_h )

norice margin added to the interpreted voltage (compared to enforced voltage)

V\_in > V\_out usually?

VTC (Voltage Transfer Characteristic) ⇐ how the V\_out levels of a gate transition between the logical low (0) and logical high (1) states with variations in the V\_input

Constraints in Processor Design

#1 - Chip area $ ⇒ limited size

large chip ⇒ high cost (no Moore’s Law)

large chip ⇒ high yield % (twice large ⇒ more than twice cost)

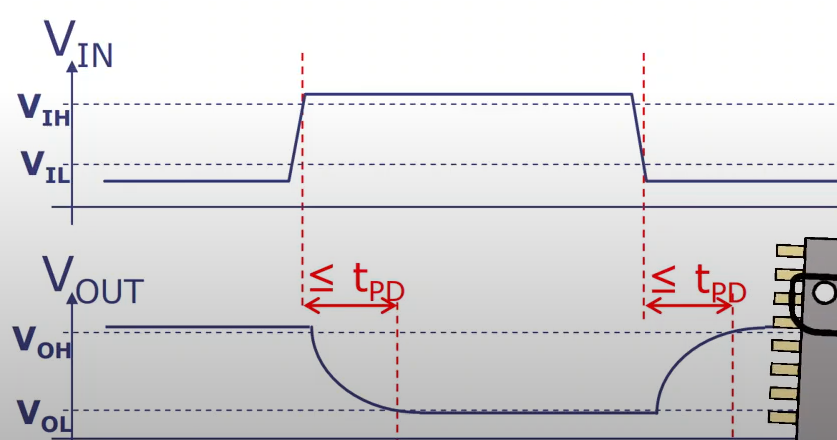
#2 - attainable clock speed ⇒ limited clock speed

more complex ISA ⇒ slow clock

combination circuit: output = f(input)

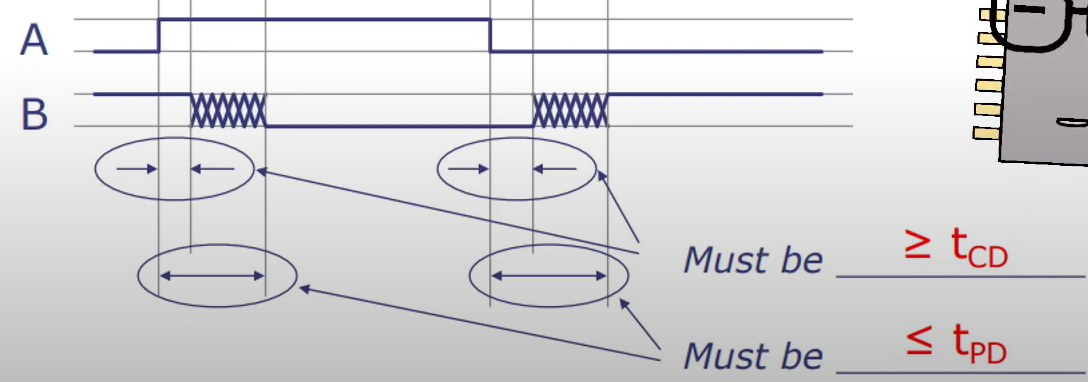
timing constraint = given dynamic input, how does output change?

propagation delay t\_pd = max time for output to be stable



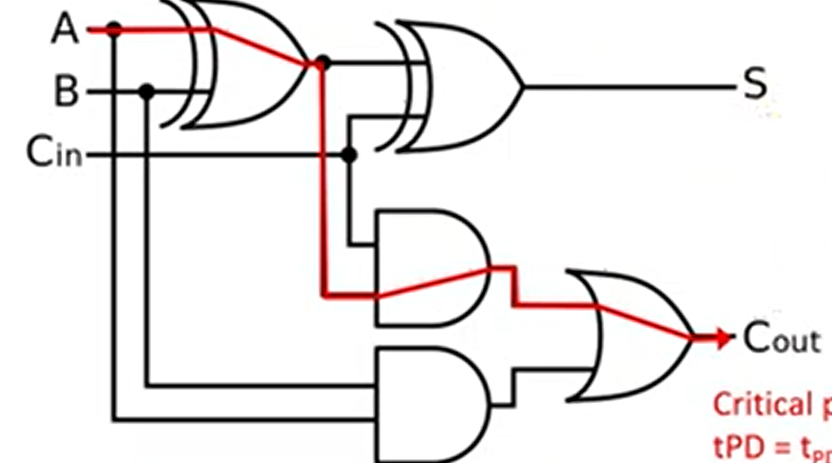
contamination delay t\_cd = min time for output to change (not stable)

give a input, V\_out will not change until t\_cd



chain of logic components has additive delay

critical path ⇒ overall propagation delay of a circuit



t\_pd = t\_pd(xor) + t\_pd(and) + t\_pd(or)

sequential circuit has memory (state): output ⇐ sequence of past inputs

states act as input to internal combinational circuit

all potential critical path through sequential circuits MUST < clock signal length

if not, either simplify logic or reduce clock speed (longer clock signal)

synchronous sequential circuit

speed ⇐ delay of longest critical path

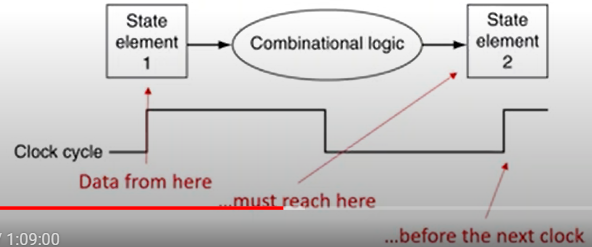
all clocks through synchronous circuit share a clock signal

t\_setup = time for new input to register in state

processing time must fit in clock cycle

after rising clock edge, t\_pd(SE1) + tpd(CL) + t\_setup(SE2) <= clock period

Otherwise, result is not registered ⇒ timing violation

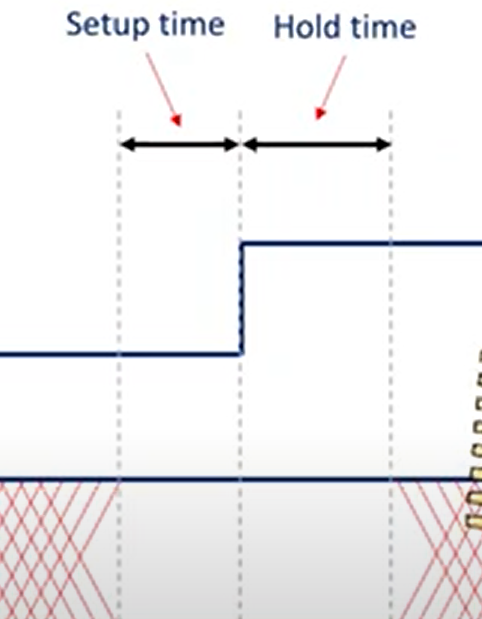


t\_hold = time to hold a data (stably) in state after clock edge

processing should not affect state too early

after rising clock edge, t\_cd(SE1) + t\_cd(CL) = guaranteed time output doesn’t change > t\_hold(SE2)

we don’t want output of CL to affect SE2



during setup and hold time, data needs to be stable

hold-time constraint is usually not hard but setup-time is hard2

Why 32-element in register files?

they are constructed in a hierarchical design ⇒ longer critical path + more t\_pd ⇒ slow clock to meet timing constraints

32-element balances # in register files + t\_pd

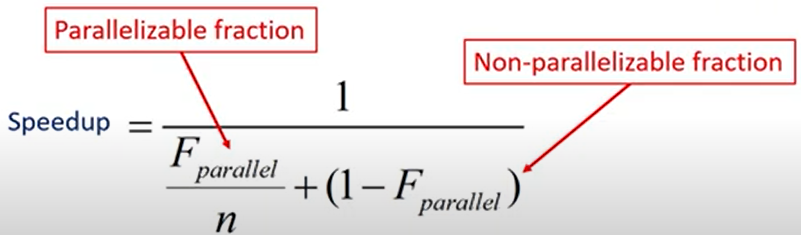
#3 - Instruction-level Parallelism (for single thread) ⇒ limited word per clock cycle

independent instructions can be processed in parallel

more ALU ⇒ more instr. executed / clock cycle BUT diminishing returns

#4 - Amdahl's Law (multi-core) ⇒ limited benefit of adding cores

some part is not parallelizable ⇒ more cores doesn’t speed up that much



For some applications, we see diminishing return

# IDEA #5 - Performance via Pipelining

RICS-V

Fetch: Request instruction fetch from memory

Decode: Instruction decode & register read

Execute: Execute operation or calculate address (ALU operations)

Memory: Request memory read or write

Writeback: Write result (either from execute or memory) back to register

why? trade off clock speed vs. complex processor

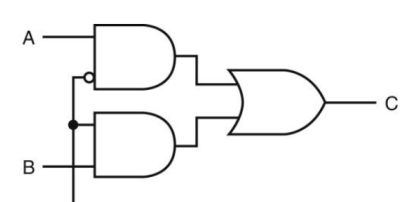
two metrics when designing a system

latency = time taken to process input and generate output

throughput = rate at which tasks are completed (number of operations or transactions that a system can handle within a given time period)

latency = t\_PD

throughput = 1/t\_PD for combinational logic

likewise

non-pipeline vs 2-stage pipeline

non-pipeline:

latency = max(F, G) + H = 20+25 = 45

throughout = 1/45

2-stage pipeline: adding registers in H to hold F & G’s output ⇒ while H is working on task\_i, F and G can work on task\_i+1

latency = max(G,F,H) \* 2 = 50

throughput = 2/50 = 1/25

k-stage pipeline

latency = k \* t\_clk

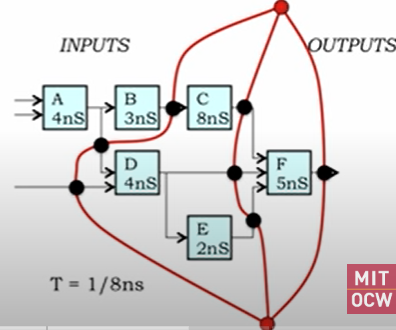
throughput = 1/t\_clk

III-formed pipelines cannot form a k-stage pipeline

because A→B→C = 1 register violates the other 2-register paths

all path must have same number of registers on the way

Pipeline Methodology



start from output, draw a line

F is the first layer, draw a line to separate it from the next layer

C D E are the second layer, draw a line to separate them from the rest

(we could have included E in the first layer, but it wouldn’t change the latency)

throughput = 1/8ns (because it’s the largest clock)

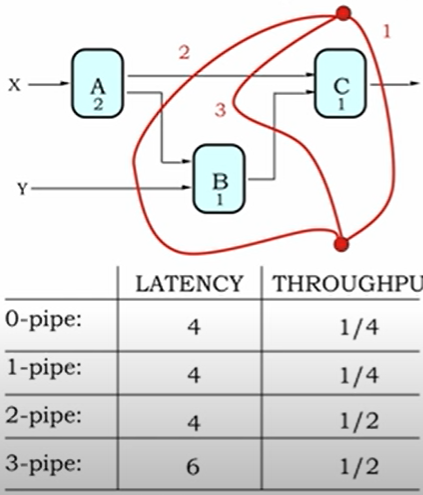
latency = k \* 8 = 3 \* 8 = 24 ns

* best division of the pipelining is to put registers on bottleneck component, which is C’s input and output

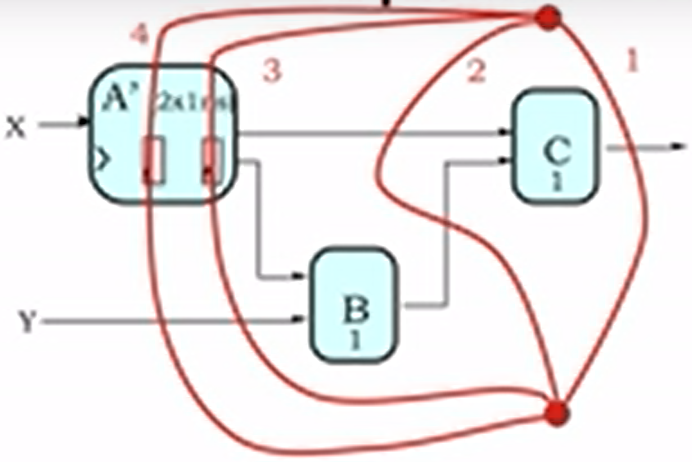
adding 1 doesn’t change the L nor T

adding pipeline 2 ⇒ higher T

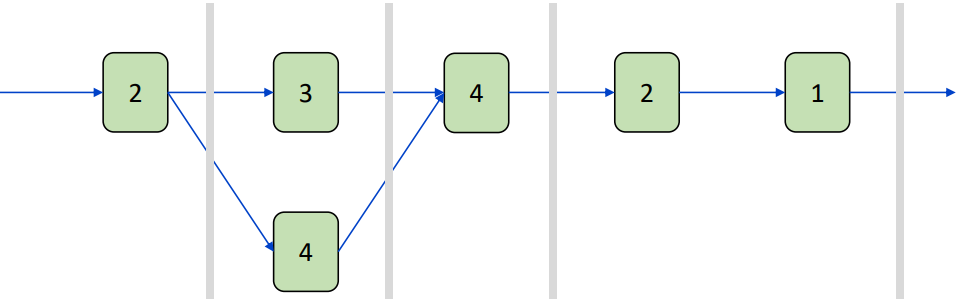
adding pipeline 3 ⇒ no change, doesn’t worth doing so



* L (pipelined) >= L (unpipelined), so we at least want to increase T by isolating the bottleneck element
* Replacing a slow element (A) with a k-pipe ⇒ may reduce clock period. In this case, L = 4ns and T = 1



Ex. L = 4 \* 4 = 16, T = 1/4



lock-step pipelining are deterministic

What if F’s latency is non-deterministic?

FIFOs

4-register ⇒ we can put 0-4 numbers inside

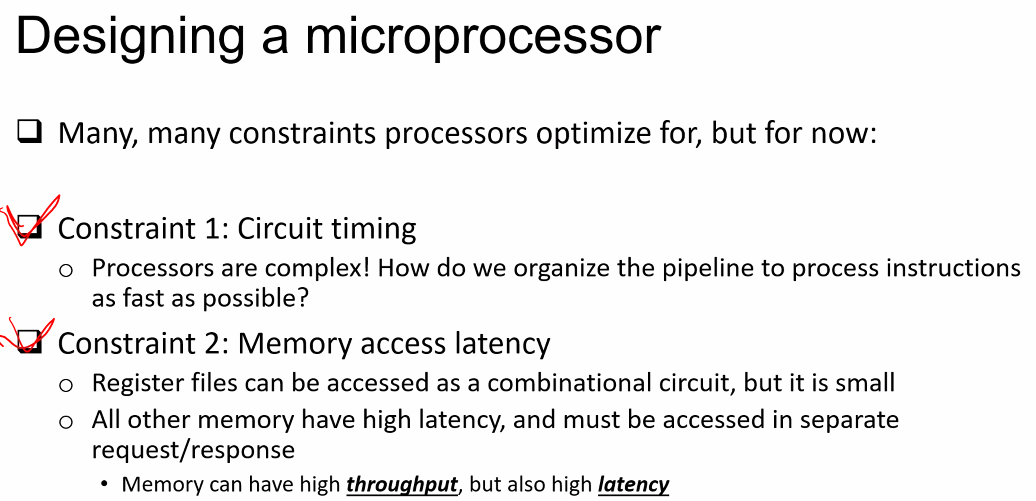
counting cycles

Aside: Little’s law

where L = # requests, =throughput, W = latency

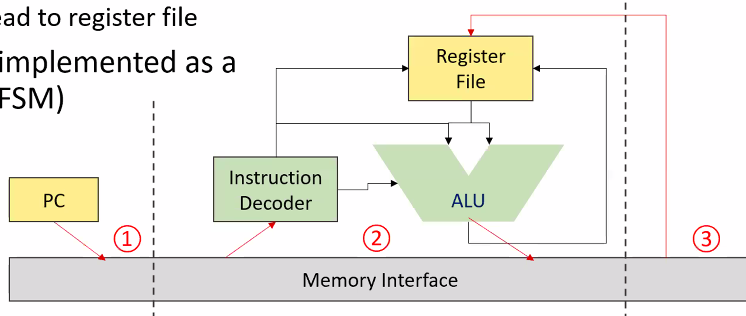
For G, L = 3

For F, W = 4



can have multiple requests → can have high throughput latency

Basic micro-architecture



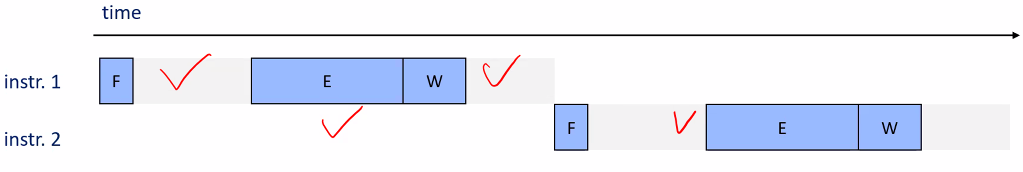
stage 1: instruction fetch

stage 2: receive instruction (only exception if ALU needs a mem read request → move to stage 3. Otherwise, go to stage 1)

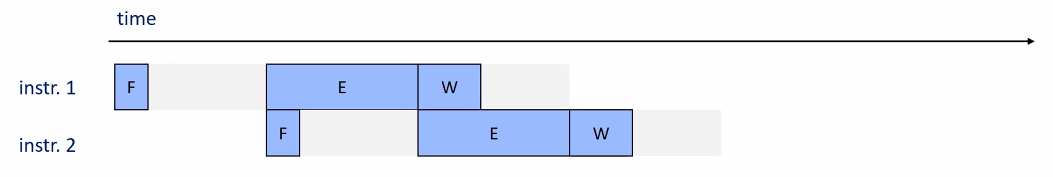
stage 3: load data from mem to register file

Will this processor be fast? no because stage 2 is too long compared to stage 1&3

before pipelining, inst.2 starts when inst. 1 finishes → 1 mem request each time

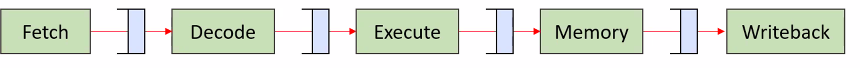


after pipelining, max 2 mem request (single mem interface cannot process two at the same time)



Balance the critical path by dividing into more stages

fetch → decode (from instruction) → execute (ALU) → memory (request mem read/write) → writeback



no single critical path which read/write to register files in one cycle

clock cycle ~ total latency / 5

(hazards ⇒ zero-initialized register will have different output than we expected)

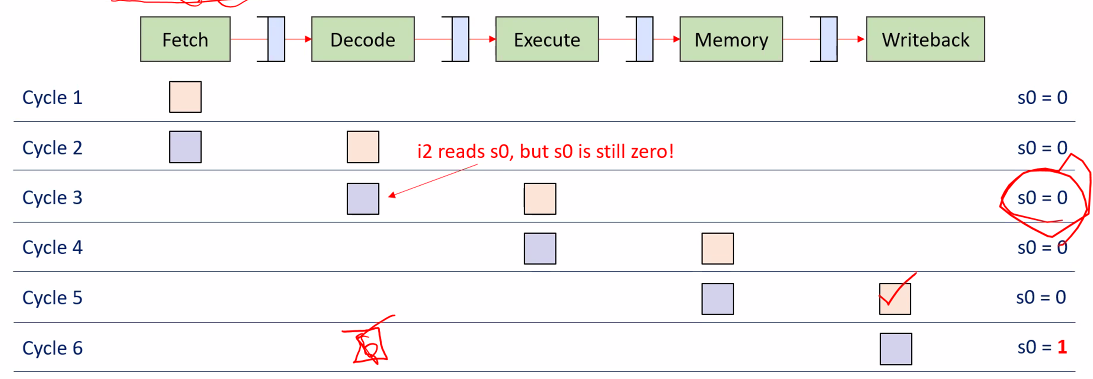
RAW (read after write) Hazard

eg.

i1: add **s0**, s1, s2

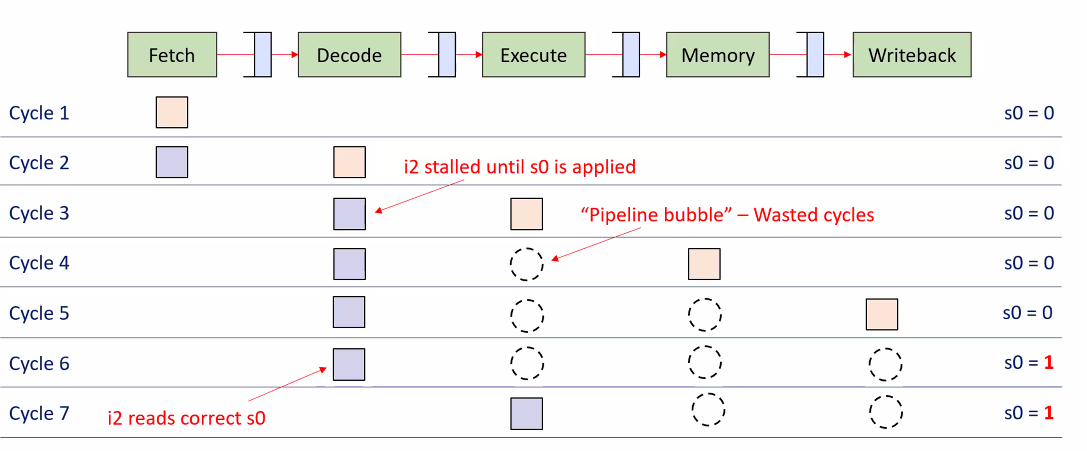
i2: add s3, **s0**, s4 (i2 read un-updated s0 which will be written by i1)

eg. i1: addi s0, zero, 1 → i2: addi s1, s0, 0



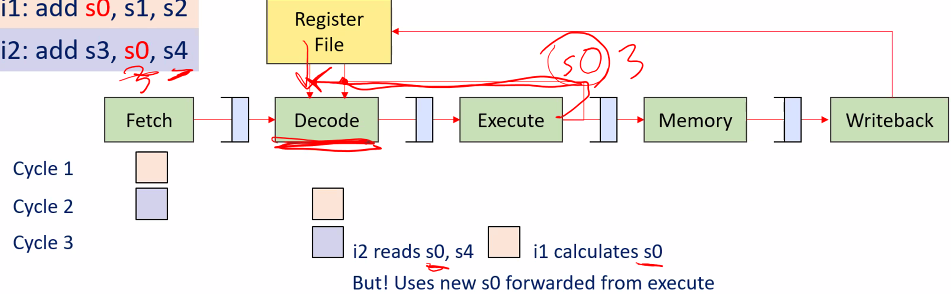
Solution# 1 - Stalling

i2 waits until i1 finishes (more bubbles (idle stages) are bad)



Solution# 2 - Forwarding

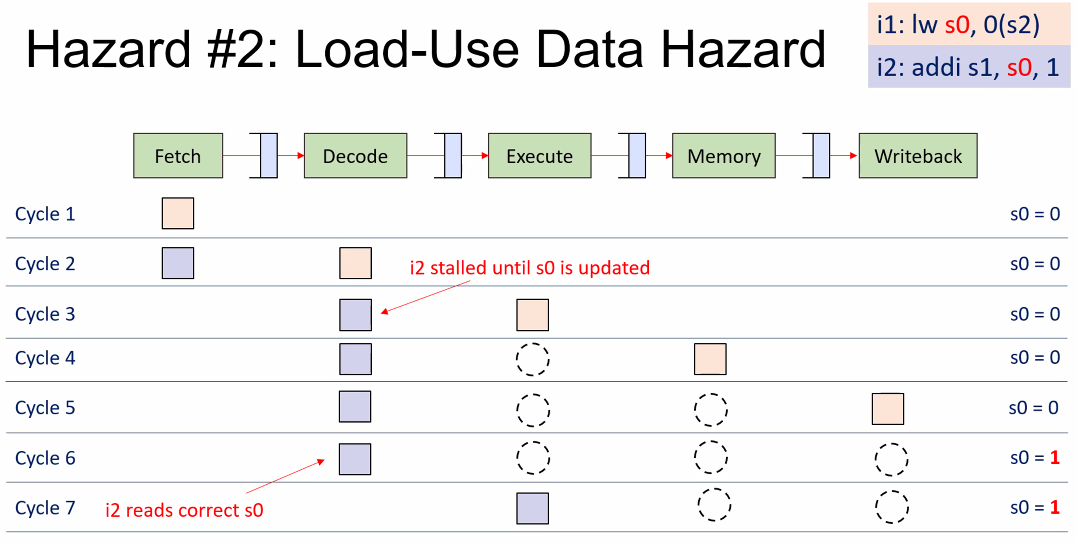
forward execution result to input of decode stage



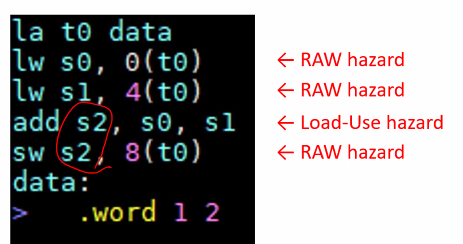
may still need stilling  
add combinational path from execute to decode

Load-Use Hazard

stilling



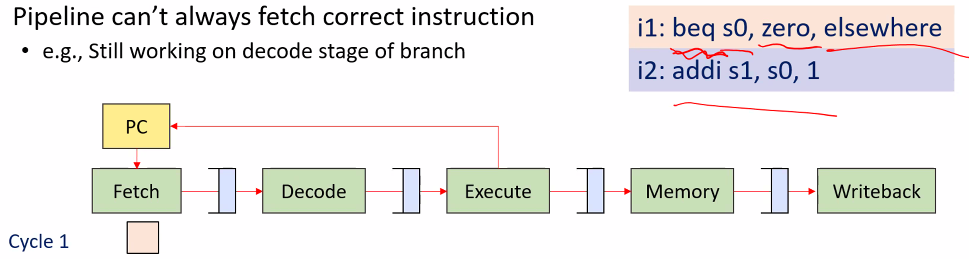
Eg.



WAW (write after write) + WAR (write after read) can be a hazard for out-of-order processor

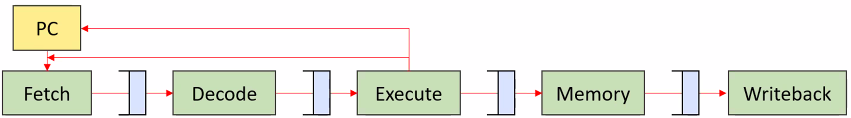
Control Hazard

branch → fetching next instruction depends on the previous branching result



Solution

branch target address can be forwarded to fetch stage (reduce 1 bubble)



decode stage can be augmented with logic to calculate branch target

⇒ imbalance pipeline ⇒ reduce performance

branch delay slot (execute next instruction any ways)

branch prediction

# IDEA #6 Performance via Prediction

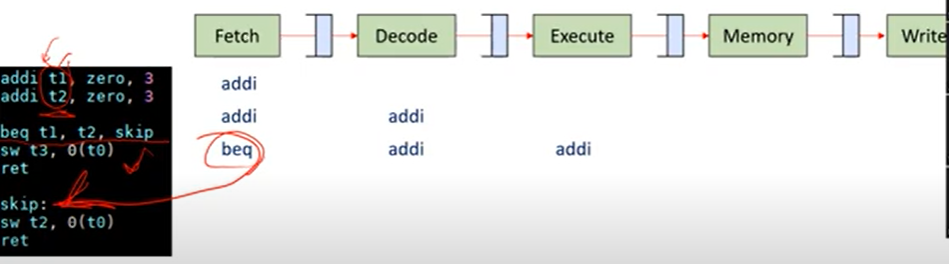
Branch Prediction

try to predict → if not correct

MUST detect mis-predict before any state change ⇐ diff to revert register writes (Write stage), memory I/O (Mem stage)

it’s possible to have load memory stage before execute

ex. simplest branch prediction – branch not taken



sw t3 beq addi addi

ret sw t3 beg addi addi

**mispredict!**

sw t2 () () beg addi

**for correct branch**

ret sw t2 () () beg

Handling mis-predictions

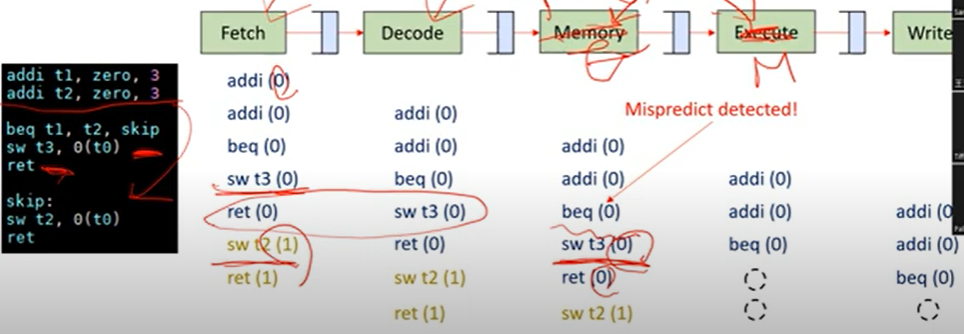
execute sends signal to all previous stages (fetch & decode) to turn them into “nop”

epoch-based

tag current instr with the epoch

mis-prediction → increment epoch and ignore the instrs with previous epoch

ex.



Static Branch Predictor

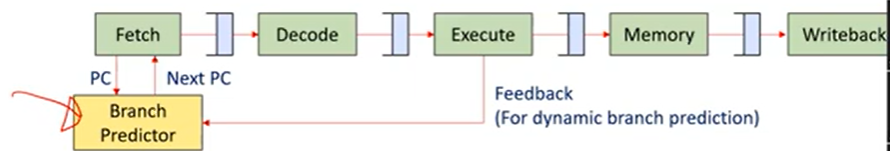
predict backward = branches taken

predict forward = branches NOT taken(ex. loop and if)

Dynamic Branch Predictor

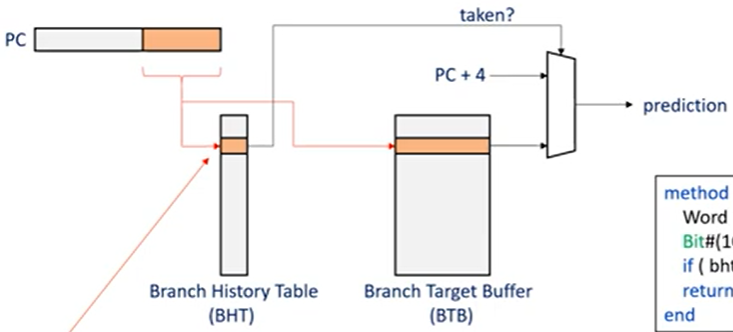
measure branch behavior → assume future behavior will behave the same

wrong → stall



Branch History Table (BHT) ⇒ if a instr will cause a branch

Branch Target Buffer (BTB) ⇒ which address this instr will jump to



space limit ⇒ usually use a relatively small table ⇒ collision

if BHT decides the taken path, we will need BTB to provide more infor

Ex. 1-bit predictor (initially will choose Not taken)

TTTTTNNNNN → TTTTTNNNNN

TNTNTNTNTN → TNTNTNTNTN

Ex. 2-bit predictor (actually taken → increase if not 11)

00 strongly not taken

01 not taken

10 taken

11 strong taken

TTTTTNNNNN → TTTTTNNNNN

TNTNTNTNTN → TNTNTNTNTN (should it be all wrong predicted??)

Evaluating approaches

profiling tools (valgrind, gproof, Linux perf)

Loop unrolling

for(i = 0..15) foo() OR for(i = 0…5) foo(); foo(); foo()

Counting Numbers

for(i = 0…cnt){

if(a[i] < 128 && b[i] < 128) lcnt++;

}

Attempt #1: loop unrolling ⇒ slightly increase performance

seems like if statements are the bottleneck (ex. comparing against 64 ⇒ 1/2 of time)

Attempt #2: branchless code by removing if

lcnt += (((a[i] - 128) >> 31) & 1) \* (((b[i] - 128) >> 31) & 1);

1/7 of time

* only one comparison in if ⇒ automatic optimization for you most of the time

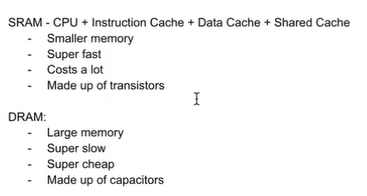
Superscalar Processing

in-order superscalar example

out-of-order

is it worth it? more simpler cores? paidoff for ML. use smaller cache if no enough

space for cache



VLIW (very long instruction word)

multiple instr packaged into a VLIW

Good for computation-intensive code + Bad for code with many dependencies/hazards

Issue #1 - Execution unit configurations change across models

Cannot be binary compatible across models!

• Unless hardware provides an abstraction layer…?

• But that would add scheduler overhead, undermining VLIW (Itanium tried a good balance)

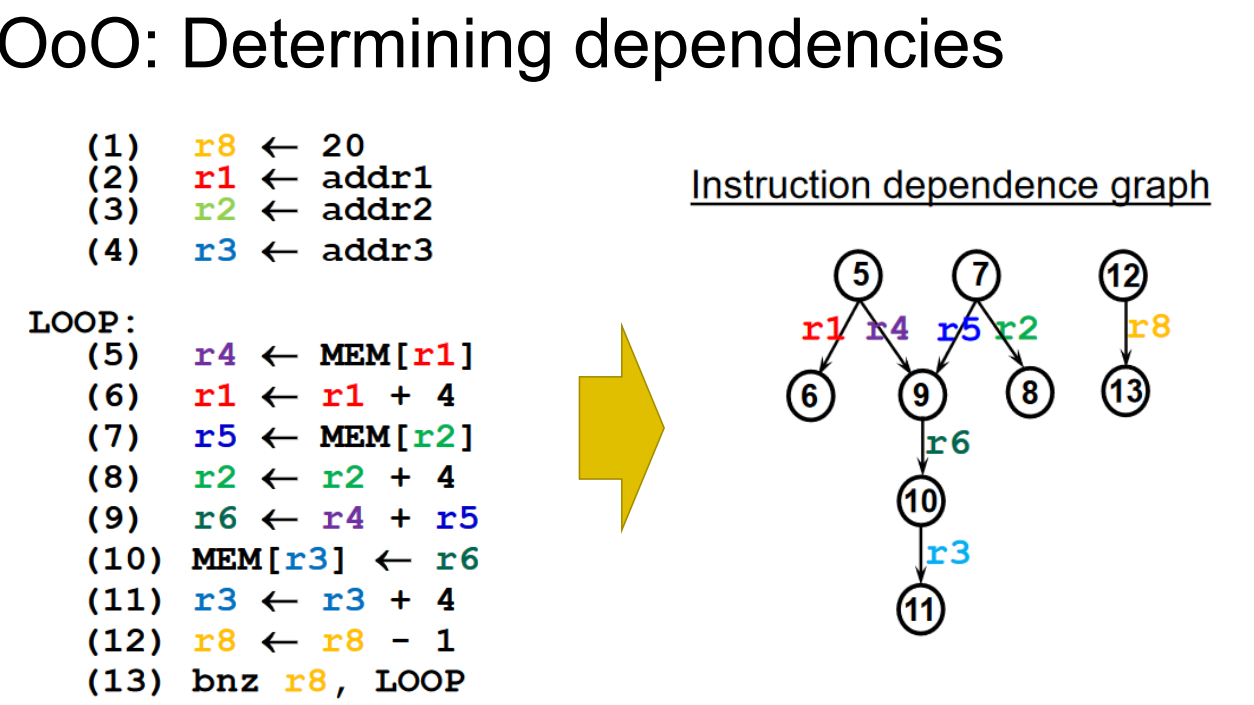
Issue #2 - Dependency/hazards difficult for compiler to manage

Too many slots end up empty (low performance, large binary)

Issue #3 - But when it works well, it works remarkably well

Out-Order Processing

Ex.



RAW dependency: 

5+9 → r4

7+9 → r5

9+10 → r6

12+13 → r8

dispatch + commit stages

Instructions wait at “reservation stations”

Forwarded to FU when ready

arithmetic can happen OoO but commit should happen in-order

Decoded instructions line up at Reorder Buffer(RoB)

Wait until execute results available

Wait until branch mispredict ruled out

Commits in order of insertion

Macro-op Fusion

Smaller number of instructions to process

reduces the number of cycles required for the same program execution

o While still maintaining RISC ISA (Also used in CISC / x86 with smaller instructions)

o Typical criticism of RISC is a larger number of generated instructions for same

program

• (More cycles to execute same program)

Cache

Method 1: Caching recently used addresses

Works because software typically has “Temporal Locality” : If a location has been accessed recently, it is likely to be accessed (reused) soon

o Method 2: Prefetching based on future pattern prediction

Works because software typically has “Spatial Locality” : If a location has been accessed recently, it is likely that nearby locations will be accessed soon

Unit of caching: “Block” or “Cache line”

Three types of misses

Compulsory misses (aka cold start misses)

Capacity misses ⇐ finite cache size

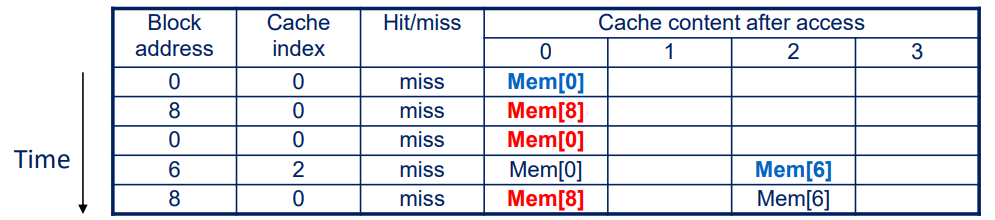
Conflict misses (aka collision misses) multiple accesses mapped to same index

⇐ competition for entries in a set

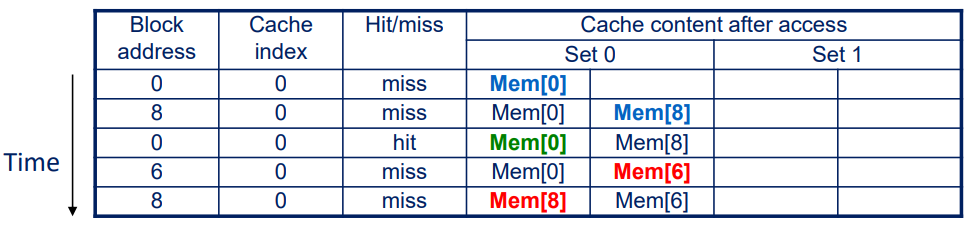
Would not occur in a fully associative cache of the same total size

Ex. caches with four elements

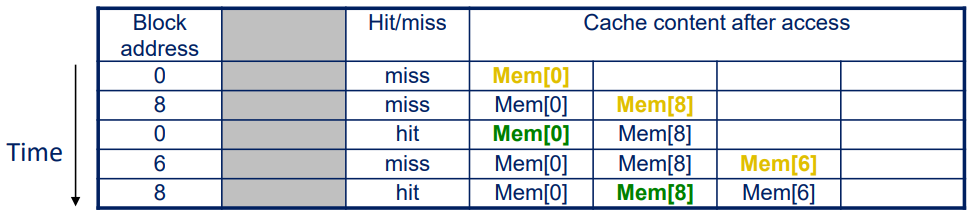
direct mapped - Cache index = address mod 4



2-way set associative (Cache index = address mod 2) Least Recently Used (LRU)



Fully associative (No more cache index!)



* LRU becomes bad if working set becomes larger than cache size

“for (i = 0 to 1025) A[i];”, if cache is 1024 elements large, every access is miss

* Large cache line size ⇒ good spatial locality + reduce size of tag mem + higher miss rate (<= fewer indices)

AMAT=Hit Time + Miss Rate×Miss Penalty where:

Hit Time is the time it takes to access the cache on a hit.

Miss Rate is the rate at which cache misses occur (1 - Hit Rate).

Miss Penalty is the time it takes to access the next level in the memory hierarchy (in this case, the main memory).

Ex.

This cache has an 50% hit rate, and a latency of 1 ns. The main memory has a latency of 20 ns. The cache design is blocking, so that we can only know whether to make the next layer cache access after we get the results back from the previous level

Given:

Hit Rate: 50% (0.5)

Miss Rate: 1 - Hit Rate = 50% (0.5)

Cache Hit Time: 1 ns

Memory (Cache) Miss Penalty: 20 ns

We have

AMAT = 1 + 0.5 \* 20 = 11 ns

Assume we have the technology to add a second layer of cache, this time with 10 ns of latency, and we measured a 40% hit rate on our benchmarks

We have

AMAT = 1 + 0.5\*10 + 0.5\*0.6\*20 = 1 + 5 + 6 = 12 ns

Summary

loading immediate number can use combination of LUI, ADDI, and shift

adding more registers ⇒

less mem access

more instr to finish one task ⇒ more time needed / clock ⇒ slow down clock speed?

more propagation delay ⇒ slow down clock speed

* ⇒ CPU Time=InstructionsPrograms\*Clock cyclesInstruction\*SecondsClock cycle
* Low instruction count ⇒ Each instruction should do more work
* Low CPI High clock speed ⇒ Each instruction should be simpler
* reduce clock speed ⇒ less accurate

RISC-V encoding

4 bytes per instruction

diff instructions have diff param (reg, immediate, …)

various fields should be encoded to consistent location

ANSWER ?? (Simplicity, ease of decoding, reduce complexity of hardware)

x86 encoding

variable-length encoding, 1 to 1 bytes encoding

complex decoding logic to translate instruction to simpler micro op

RISC-V

why not shift left? because shift right is the same logic as the shift left.

RISC

simplicity, regularity, and efficiency in instruction execution ⇒ fixed instruction formats + limited addressing modes + straightforward immediate value handling.

CISC

complex + variable-length instructions ⇒ powerful instructions + minimize # instructions for a task + slow down the clock

arithmetic + load/store

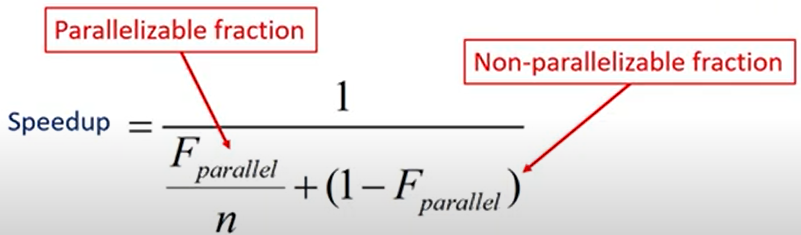
#3 - Instruction-level Parallelism (for single thread) ⇒ limited word per clock cycle

independent instructions can be processed in parallel

more ALU ⇒ more instr. executed / clock cycle BUT diminishing returns

#4 - Amdahl's Law (multi-core) ⇒ limited benefit of adding cores

some part is not parallelizable ⇒ more cores doesn’t speed up that much



RICS-V

Fetch: Request instruction fetch from memory

Decode: Instruction decode & register read

Execute: Execute operation or calculate address (ALU operations)

Memory: Request memory read or write

Writeback: Write result (either from execute or memory) back to register

why? trade off clock speed vs. complex processor

non-pipeline vs 2-stage pipeline

non-pipeline:

latency = max(F, G) + H = 20+25 = 45

throughout = 1/45

2-stage pipeline: adding registers in H to hold F & G’s output ⇒ while H is working on task\_i, F and G can work on task\_i+1

latency = max(G,F,H) \* 2 = 50

throughput = 2/50 = 1/25

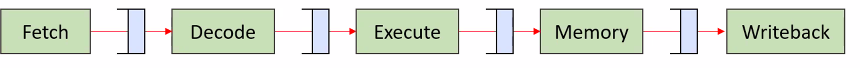
k-stage pipeline

latency = k \* t\_clk

throughput = 1/t\_clk

Balance the critical path by dividing into more stages

fetch → decode (from instruction) → execute (ALU) → memory (request mem read/write) → writeback



no single critical path which read/write to register files in one cycle

clock cycle ~ total latency / 5

(hazards ⇒ zero-initialized register will have different output than we expected)

switching mem with exe ⇒

reduce number of instr

reduce clock speed

need runtime benchmarking to be sure

RAW (read after write) Hazard

eg.

i1: add **s0**, s1, s2

i2: add s3, **s0**, s4 (i2 read un-updated s0 which will be written by i1)

Solution# 1 - Stalling - i2 waits until i1 finishes (more bubbles (idle stages) are bad)

Solution# 2 - Forwarding - forward execution result to input of decode stage

may still need stilling  
add combinational path from execute to decode

Load-Use Hazard

stilling

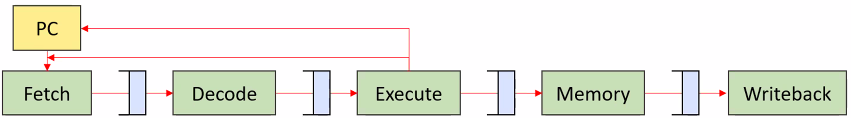
WAW (write after write) + WAR (write after read) can be a hazard for out-of-order processor

Control Hazard

branch → fetching next instruction depends on the previous branching result

Solution

branch target address can be forwarded to fetch stage (reduce 1 bubble)



decode stage can be augmented with logic to calculate branch target

⇒ imbalance pipeline ⇒ reduce performance

branch delay slot (execute next instruction any ways)

branch prediction

try to predict → if not correct

MUST detect mis-predict before any state change ⇐ diff to revert register writes (Write stage), memory I/O (Mem stage)

Ex. 1-bit predictor (initially will choose Not taken)

TTTTTNNNNN → TTTTTNNNNN

TNTNTNTNTN → TNTNTNTNTN

Ex. 2-bit predictor (actually taken → increase if not 11)

00 strongly not taken

01 not taken

10 taken

11 strong taken

TTTTTNNNNN → TTTTTNNNNN

TNTNTNTNTN → TNTNTNTNTN (should it be all wrong predicted??)

Superscalar Processing

out-of-order

is it worth it? more simpler cores? paidoff for ML. use smaller cache if no enough space for cache

VLIW (very long instruction word)

multiple instr packaged into a VLIW

does not change the ISA + Complicates hardware in charge of detecting dependencies

Good for computation-intensive code + Bad for code with many dependencies/hazards

Issue #1 - Execution unit configurations change across models

Cannot be binary compatible across models! unlike Macro-op Fusion

• Unless hardware provides an abstraction layer…?

• But that would add scheduler overhead, undermining VLIW (Itanium tried a good balance)

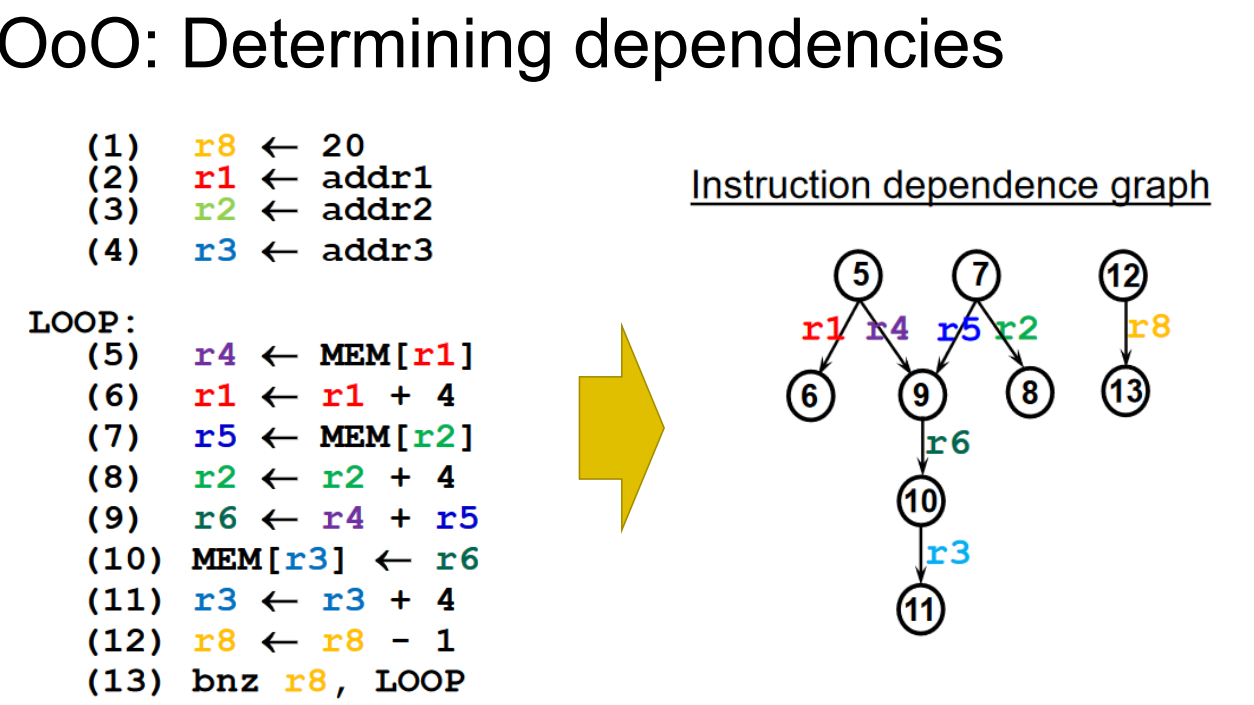
Issue #2 - Dependency/hazards difficult for compiler to manage

Too many slots end up empty (low performance, large binary)

Issue #3 - But when it works well, it works remarkably well

Out-Order Processing

Ex.



RAW dependency: 

5+9 → r4

7+9 → r5

9+10 → r6

12+13 → r8

dispatch + commit stages

Instructions wait at “reservation stations”

Forwarded to FU when ready

arithmetic can happen OoO but commit should happen in-order

Decoded instructions line up at Reorder Buffer(RoB)

Wait until execute results available

Wait until branch mispredict ruled out

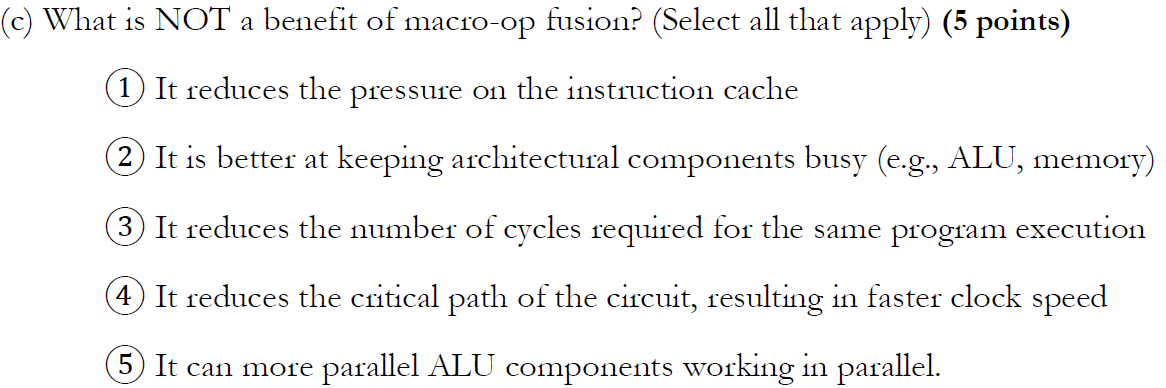
Commits in order of insertion

Macro-op Fusion

if two instr are outputting different register, we cannot fuse them together because we only have one writeback.

only happen in the decode stage.

reduces the number of cycles required for the same program execution



(2)(3) are the benefits of macro-op fusion

Cache

Method 1: Caching recently used addresses

“Temporal Locality” - most recently used one, will be most likely to be used soon

o Method 2: Prefetching based on future pattern prediction

“Spatial Locality” - recently accessed one, its nearby location will be used soon

Unit of caching: “Block” or “Cache line”

Why can’t we have a single SRAM cache with the size of all caches combined, instead of a hierarchy? because such a cache would have lower AMAT

Three types of misses

Compulsory misses (aka cold start misses)

Capacity misses ⇐ finite cache size

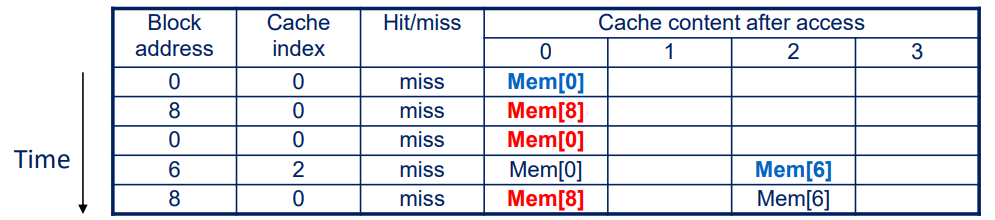
Conflict misses (aka collision misses) multiple accesses mapped to same index

⇐ competition for entries in a set

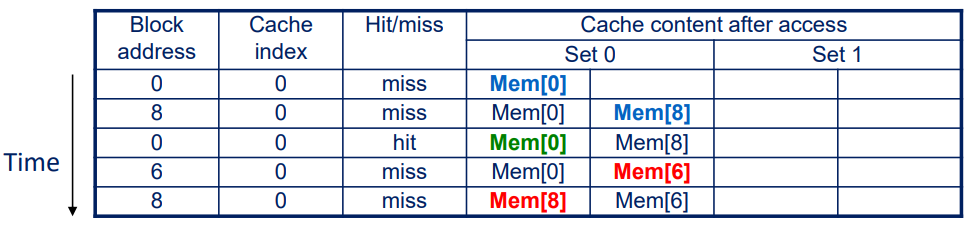
Would not occur in a fully associative cache of the same total size

Ex. caches with four elements

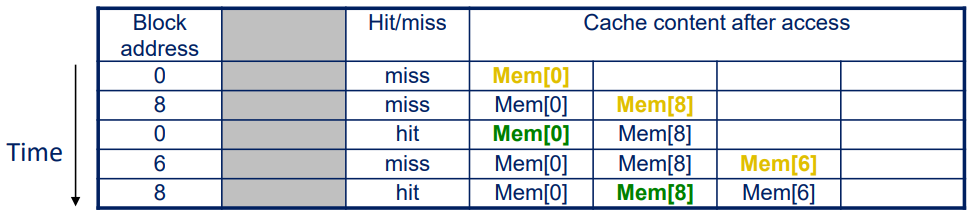
direct mapped - Cache index = address mod 4



2-way set associative (Cache index = address mod 2) Least Recently Used (LRU)



Fully associative (No more cache index!)



* LRU becomes bad if working set becomes larger than cache size

“for (i = 0 to 1025) A[i];”, if cache is 1024 elements large, every access is miss

* Large cache line size ⇒ good spatial locality + reduce size of tag mem + higher miss rate (<= fewer indices)

AMAT=Hit Time + Miss Rate×Miss Penalty where:

Hit Time = time to access the cache on a hit.

Miss Rate = rate at which cache misses occur = (1 - Hit Rate).

Miss Penalty = time to access the next level in the memory hierarchy

Ex.

This cache has an 50% hit rate, and a latency of 1 ns. The main memory has a latency of 20 ns. The cache design is blocking, so that we can only know whether to make the next layer cache access after we get the results back from the previous level

Given:

Hit Rate: 50% (0.5)

Miss Rate: 1 - Hit Rate = 50% (0.5)

Cache Hit Time: 1 ns

Memory (Cache) Miss Penalty: 20 ns

We have

AMAT = 1 + 0.5 \* 20 = 11 ns

Assume we have the technology to add a second layer of cache, this time with 10 ns of latency, and we measured a 40% hit rate on our benchmarks

We have

AMAT = 1 + 0.5\*10 + 0.5\*0.6\*20 = 1 + 5 + 6 = 12 ns

# OS Support History

History:

Not hardware abstraction

Each software had to handle each possible video, sound, etc hardware

Software failure -> System crash!

User software had all access to hardware, including OS files on disk

Only one software running at a time!

CPM (single-tasking) sort several address to fill pointers (to actual OS code and data, which located on top of the available memory capacity)

benefit:

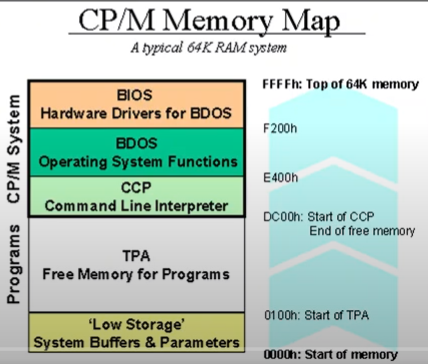
OS code is in the same location regardless changes of program/OS/storage driver

user program will always start from the same location (100x bits)

When done execution, simply returns to OS

Software has exclusive access to machine OS is effectively just like a library – DOS was very similar

BUT only works if only running one program at a time



Multi-tasking

We cannot use absolute addressing any more for jumps and data referencing

No longer simple address model with assumed exclusive access to memory

**Modern OS support user process isolation via “privilege levels”**

**private address space** (no other user process can access OS mem)

**schedules processes**

Each process is given a fraction of CPU time

A process cannot use more CPU time than allowed

system services (e.g., access files or network sockets) via system calls

abstraction: no need to show hardware details

## Privilege Level RISC-V

three (or more) formally defined levels

o **Machine level**, full access to all hardware after initial boot

o Hypervisor level – For virtualization. Recently formally defined! (2022)

o Supervisor level – For operating systems

o **User level** – For applications

Special register, “mstatus” (for “machine status”)

mstatus stores the privilege level of the current process

Writing a new value to it can change the privilege level

only machine mode processes are allowed to write to it

OS completes kernel process before starts user mode (never jump back to higher level)

Special ISA instructions to access the special registers

o One of many “Control Status Register” (CSR)

o csrr, csrw instructions, only allowed in machine mode

## Interrupts and exceptions to transition from user to supervisor mode

Exceptions need to be handled by OS kernel (I→H)

stop current process at I\_i and completes I\_i-1

saves PC of I\_i and why this happen

enables supervisor mode + disable interrupts + transfer control to exception handler PC

after exception handler

processor returns to I\_i or OS abort process

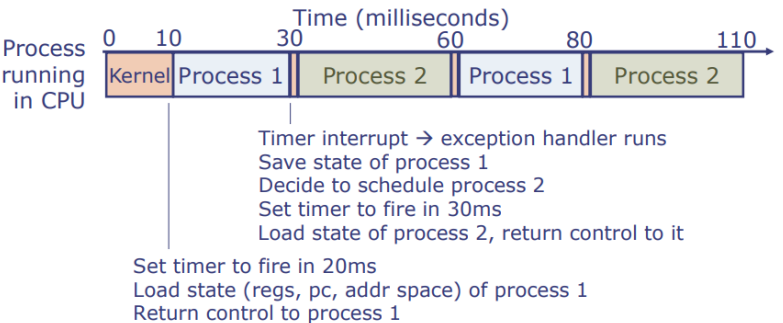
* process knows a table of predefined exceptions and how to handle them
* Only hardware involved in process (consult register, reads table, jump to the correct handler)

Interrupts caused by outside, exceptions caused by itself

Exception use #1 - CPU scheduling

OS kernel schedules multiple processes into 1 CPU

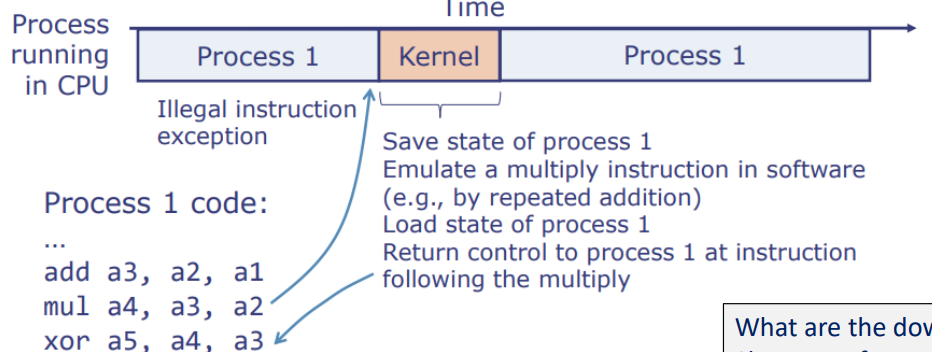
Timer interrupts to limit time allowed for each process



Exception use #2 - Emulating Unsupported Instructions

some instruction is not implemented → abort or emulate

Ex. mul x1, x2, x3 is an instruction in the RISC-V ‘M’ extension, but not supported in RISC-V machine



downside - Slower performance compared to RISC-V “M” implementation!

Exception use #3 - System Calls

User process has NO access to raw hardware resources

User process communicates with the OS via system calls,...

syscall instruction → a machine-mode exception to handle request

Exception Detail in RISC-V

control and status registers (CSRs) accessible by higher-privilege process

o mepc: PC of instruction that caused exception

o mcause: cause of the exception (interrupt, illegal instr, etc.)

o mtvec: address of the exception handler (table vector?)

o mstatus: status bits (privilege mode, interrupts enabled, etc.)

RISC-V also provides privileged instructions, e.g.,

o csrr and csrw to read/write CSRs

o mret to return from the exception handler to the process

o Trying to execute these instructions from user mode causes an exception. normal processes cannot take over the system

System call details for RISC-V

ecall instruction causes an exception, sets mcause CSR to a particular value

Application Binary Interface (ABI) convention defines how process and kernel pass arguments and results

o Typically, similar conventions as a function call:

o System call number in a7

o Other arguments in a0-a6

o Results in a0-a1 (or in memory)

o All registers are preserved (treated as callee-saved) Why is this?

interrupts usually invoked when user process is not expecting

reserving registers will ensure we don’t have lost values

——————————

Context switching (Exception use #1 - CPU scheduling)

Reason:

simple jumping doesn’t ensure correctness ⇐ all registers are callee-saved and can be changed by other user process

still in processor level. it’s just executing instructions

Process Control Block (PCB)

stores Context information (process ID, context state (register values, etc), meta-information for scheduling control)

**OS software (not the processor hardware)** is responsible for context switching, including

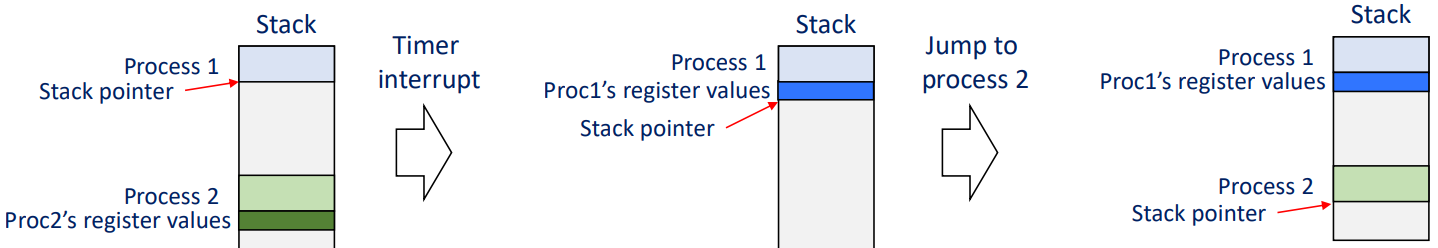
o Storing the current context to the appropriate PCB

o Deciding which process to execute (and for how long)

o Loading the next context from the PCB to the hardware registers

o Resuming the next process

“Resuming” because it is currently suspended while the current process was executing



* **No isolation between processes** (still share same mem space)
* x86 supports hardware handling of context switching operations
  + BUT most OS don’t use it.
* x86 doesn’t provide a way to explicitly switch to user level
  + Instead, we use code pretending to return from an interrupt, back into user level
* System boot process via Firmware (located in address 0)
  + when power on, CPU executes the firmware to load a small “bootloader”
  + Bootloader loads the actual OS kernel from storage to memory and transfers control
* Why bootloader?
  + Regardless of BIOS or UEFI, bootloader locates/loads/executes the OS kernel
  + BIOS (Basic Input/Output System) treated the first sector (512 Bytes) of a storage medium specially (MBR, “Master Boot Record”)
  + UEFI (Unified Extensible Firmware Interface) doesn’t use MBR

——————————

## Virtual memory to provide isolation between processes

Virtual address

o Address generated by a process (given to LW, SW, etc…)

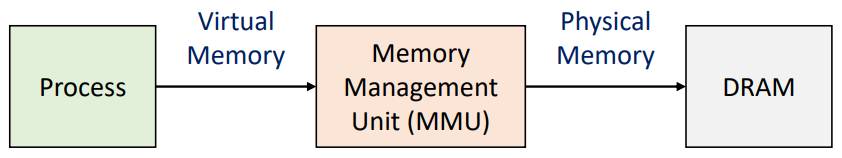
o Specific to a process’s private address space

Physical address

o Actual address on the physical DRAM module

o Only managed/visible to the operating system

MMU = hardware memory management unit (in the CPU, configured by OS)



Each process can only interact with memory space via virtual memory

o OS controls which physical memory is accessible by each process

Virtual memory also supports demand paging

o Physical memory runs out → space is freed by moving some data to storage

o This can be done transparently by the MMU

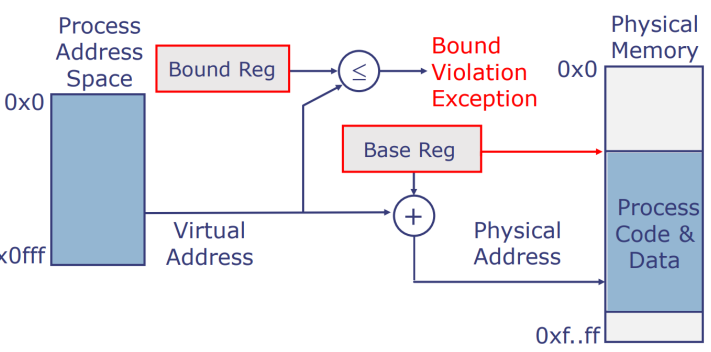
Old solution - Segmentation

program’s data is allocated in a contiguous segment of physical memory

Physical address = Virtual Address + Segment Base

Bound register provides safety and isolation

Base and Bound registers only accessible in supervisor mode (not user program)



Downsides - contiguous memory space ⇒ memory fragmentation

o Small gaps between allocated process segments ⇒ wasted mem

o New processes cannot be allocated that space, because we don’t know how much space it will need

o Multiple segment registers sets (code, stack, etc) helped a bit, but not a

fundamental solution

o Sometimes required costly defragmentation/compaction to collect free space

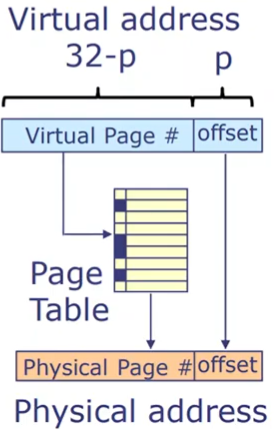
Modern solution - Paged V mem vis OS and hardware

Physical memory is divided into fixed-size units called pages (4KB usually)

Physical page number = Mem[PT base + virtual page number]

Physical address = Physical page number + offset

Virtual address = <virtual page offset, page internal offset>

size of V mem <= size of physical address 

Use a per-process page table to translate virtual page offset to physical page offset

**translation done in hardware**

Page Table Entry (PTE) = metadata flags + physical address

more flags ⇒ less space for actual address offset

“read only” flag = write protected by OS

program code often write protected

* Paging store pages non-contiguously ⇒ reduce fragmentation issue + can be mapped to share some physical pages (different V address)
* Page table is larger and requires more space

Each cache miss ⇒ 2 physical mem access

solution = TLB (Translation Lookaside Buffer) use some on-chip memory to cache virtual-to-physical address translations

o TLB hit -> Immediate translation

o TLB miss -> Must walk page table and populate TLB for later

Size of table per process + multiple processes ⇒ large mem for page table

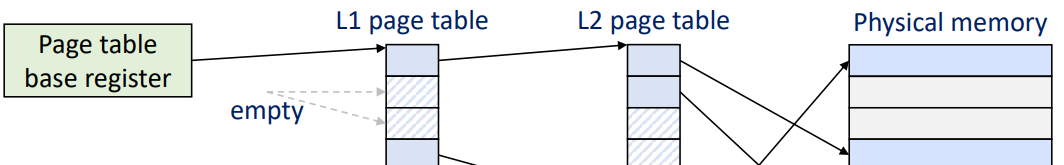
bad solution - large page sizes ⇒ waste mem + fragmentation

solution - hierarchical page tables ⇐ most V mem is not used

virtual address = **<L1 page #, L2 page #..., offset>**

entries only created when accessed

only L1 need to be pre-allocated. L2 only created when needed



* OS must be aware of the hierarchy.
* Multi-level page tables ⇒ TLB miss need >2 mem access

OS can learn which is implemented by reading the CSR “satp” (Supervisor Address Translation and Protection)

Page table base register is a CSR, “sptbr”(Supervisor Page Table Base Register)

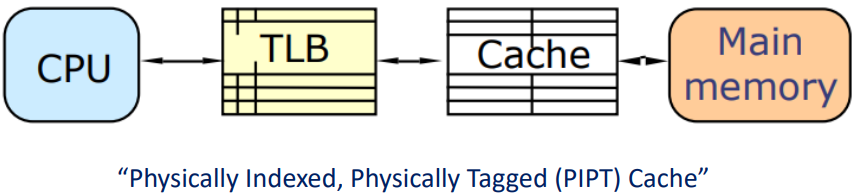
Multiprocessing Context Switching & Caching

add page table base register to the context (to switch correctly)

Cache #1 PIPT (physically indexed, physically tagged)

Based on physical address ⇒ simple + handle shared memory between processes automatically

not knowing physical address to cache + TLB miss still costly⇒ slow



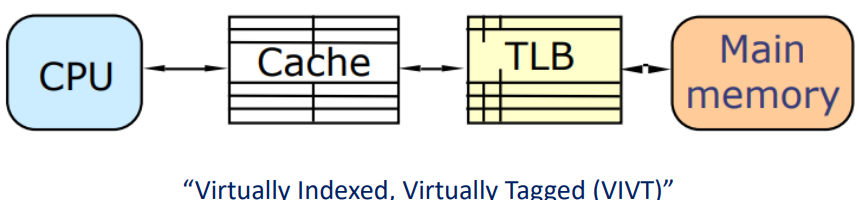
Cache #2 VIVT (virtually indexed, virtually tagged)

based on virtual add ⇒ fast cache hit

**Homonyms problem ⇒ incorrect**

different processes can and will be using the same virtual memory mapped to different physical locations

Synonyms problem (same physical add cached multiple times) ⇒ costly



Solution #1 - TLB flushed after every context switch ⇒ cold misses

Solution #2 - adding “Address Space ID” to virtual address

A small, process-specific **ASID** (Address Space ID) ⇒ fast

BUT need additional hardware+OS support

Limit number of concurrent processes

Cache #3 - VIPT (Virtually-Indexed, Physically Tagged)

mixed virtual and physical addressing

32-bit BYTE address = 28 tag bits + 2 index bits + 2 offset bits

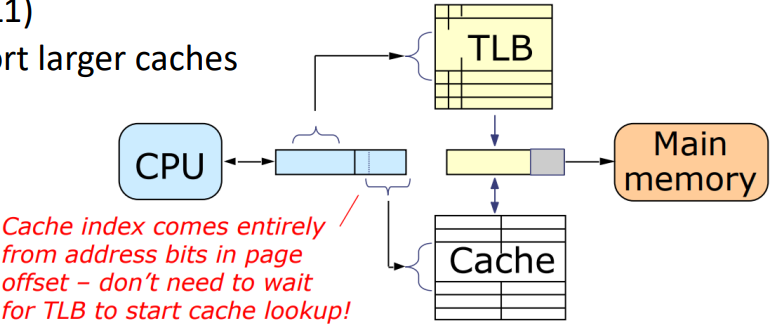
cache lookup using page offset ⇒ **(cache index length ≤ page offset length** ⇒ TLB and cache lookups in parallel)

Only works for small caches

Can increase associativity to support larger caches

TLB lookup using virtual bits

if physical bit matches from both lookup ⇒ we can process (otherwise, it’s other process’s physical address)



Solution of Synonyms

cache ensures same physical address is not cached twice

* + **L2, L3… caches use PIPT because L1 uses VIPT (virtual mem issues are addressed)**

Demand Paging during TLB

Why

Some pages are backed up to secondary storage, freeing DRAM for more pages

“Swap space” on storage used to “swap out” less-used pages

Provides illusion of very large memory capacity

PTE (Page table entry) needs additional information to handle this

Two additional flags, “Resident”, and “Dirty”

Effectively, DRAM has become a cache for disk

**Write-through writes to the backing storage** (disk in this case)

simple but performance limited by disk bandwidth

**Write back** only write to disk if the in-memory resident page is evicted, and if its contents have changed since the last time it was written

**need a “dirty” flag**, just like write back caches!

Page eviction policy

when memory runs out. Some form of LRU

Page number can now mean either offset in memory or in storage

Swapping pages needs both OS and hardware support

page swapped out and need access to it ⇒ swap back in

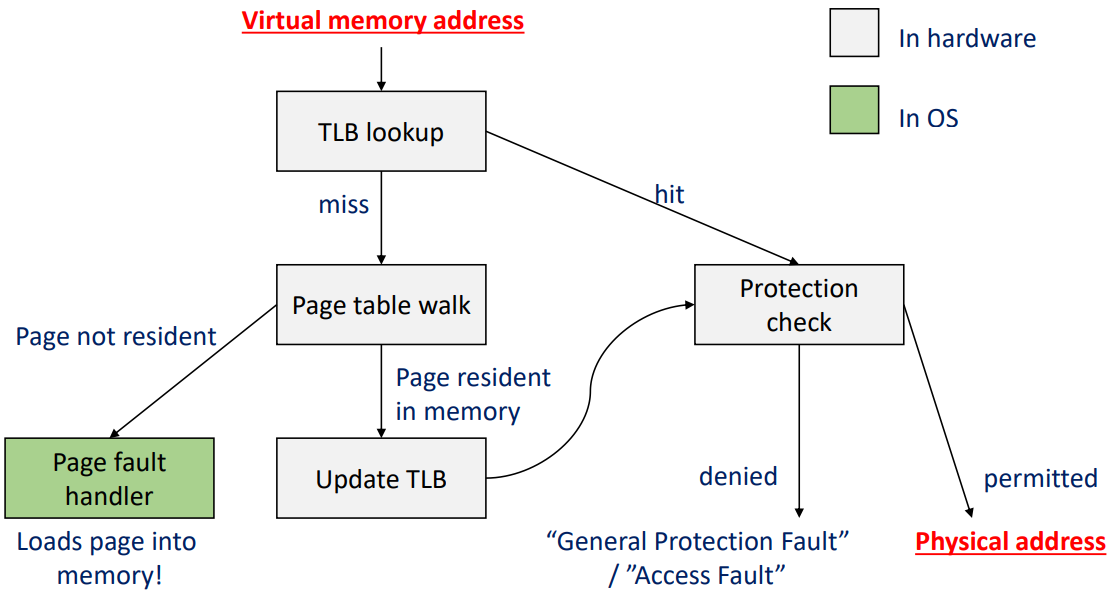
OS software handles ⇐ many logic for handling protocols,

swap page is depleted ⇒ error message

VM access to non-resident page ⇒ hardware exception Page Fault

MMU checks the page table if the “Resident” flag is set → exception

The OS page fault handler is invoked (copy page from storage to memory → set resident flag)



Page Table Flags

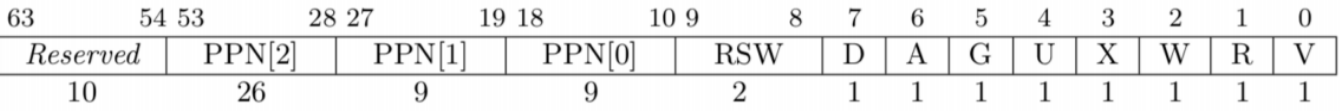
“Executable (X)”, “readable (R)”, “user mode accessible (U)”

• Extension of “read-only” flag, assigns restrictions on page access and raises an exception when violated

• Memory is often divided into kernelspace and userspace

“Was accessed (A)”, …

• For better LRU implementation



Load-on-demand strategy

Copy-on-write optimization

fork() creates a copy of process mem. ONLY copies only page table, pointing to the same physical address

make all pages read-only

When write by new/original → raise access fault → make copy

On-demand storage access

When executing a program, don’t load all of the executable file

create a page table and mark most of it non-resident

access raise page faults → read from storage

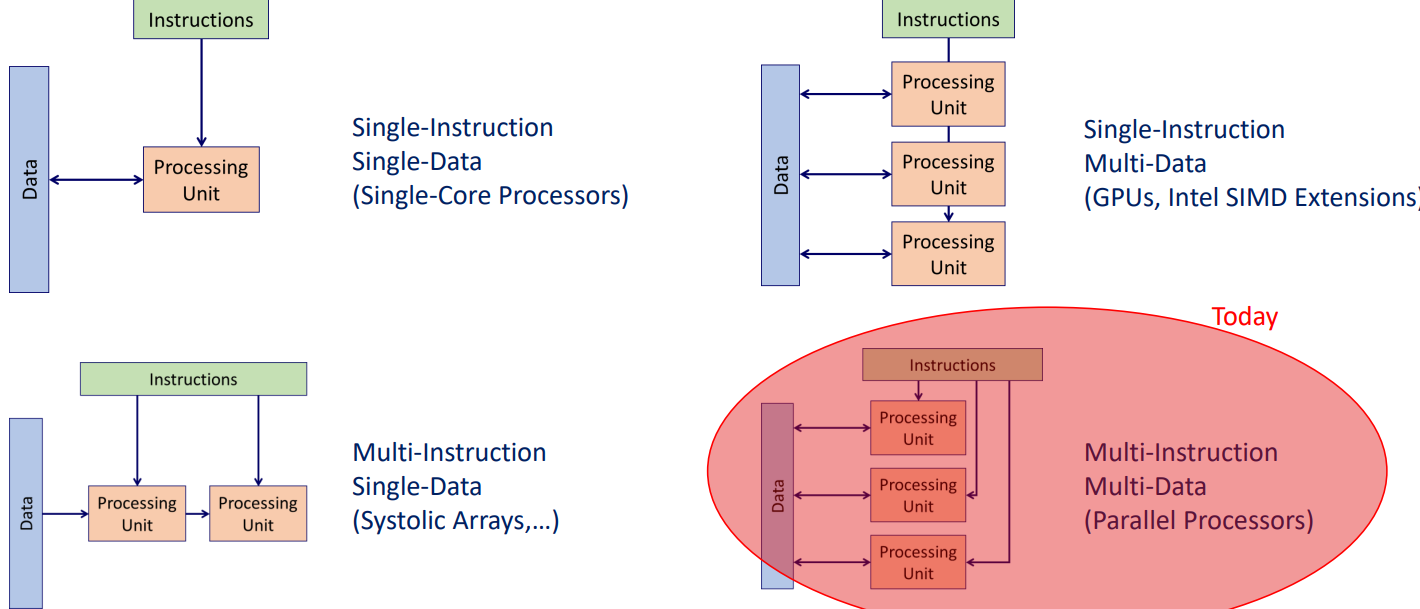
Ex. mmap() returns a pointer, to which an area in storage is mapped

No data read or copied initially, only page table entries marked nonresident

Read/write to the mmap’d pointer when raises exceptions

* If the page size is too large and access patterns are not friendly,
* most of the loaded page may never be accessed (Internal fragmentation) ⇒ Not an efficient use of memory capacity

# Multiprocessing and Parallelism



MIMD (Multi-Instruction Multi-Data)

Shared memory multiprocessor

Hardware provides single physical address space for all processors

Synchronize shared variables using locks

Memory access time

UMA (uniform) vs. NUMA (nonuniform)

SMP: Symmetric multiprocessor

The processors in the system are identical, and are treated equally (cores compete to become boot core)

Typical chip-multiprocessor (“multicore”) consumer computers

## Memory issue with multiprocessing

## Issue #1 - Cache coherency (The two CPU example)

Write after read (A read x, B write x) happens in write-through/back

Read to each address must return the most recent value

Complex and difficult with many processors

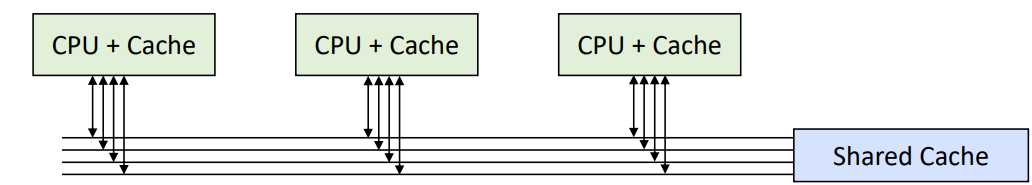
All writes must be visible at some point, and in proper order

solution #1 - broadcasting all writes to all cores (undermines the purpose of caches)

solution #2 - privately cache writes without broadcasting, whenever possible

Background

bus interconnect a shared bundle of wires

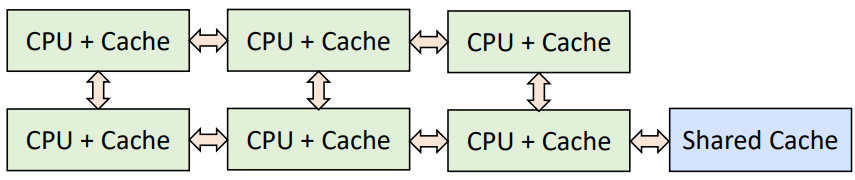


All data transfers are broadcast (one bundle)

Only one entity may be transmitting at any given clock cycle

multiple entities ⇒ “bus arbiter” entity must assign which master can write at a given cycle

Mesh interconnect



Each core acts as a network switch (several bundles)

Higher aggregate bandwidth

Bus: 1 message/cycle

Mesh: Potentially as many messages as there are links

better scalability with more cores

variable cycles of latency

More transistors to implement, compared to bus

To keep multiple cache coherent,

If a cache line is only read, many caches can have a copy

If a cache line is written to, only one cache at a time may have a copy

implementation #1 - “Snooping-based”

All cores listen to the traffic on the memory bus

**Before writing to a cache line, must broadcast its intention**

All other caches must invalidate its own copies

Algorithm variants exist to make this work effectively (MSI, MSIE, …)

**many writes will cause issue (dirty instance only exists in one place)**

MSI coherence

Each cache line can exist in one of three states

Modified ‘M’ : Dirty line. Only one copy exists across cores.

Shared ‘S’ : Unmodified read-only ⇒ multiple copies.

Invalid ‘I’ : Cache is no longer valid

Issue - only one can broadcast per cycle

implementation #2 - “Directory-based”

assign subsets of main memory to each core

write ⇒ directory sends P2P invalidation msg

(Pros) Scalable ⇐ multiple MSI-like protocol can run at each core

(Cons) Higher latency (multi-cycle request to directory)

(Cons) More memory requirement (directory data structure)

Basic implementation: A bit vector per cache line

P bits per cache line for P cores

1 additional dirty bit

Performance Issues of Cache Coherency

diff memory locations, written to by diff cores, mapped to same cache line

solution = store often-written data in local variables

synchronization with mutex (mutual exclusive lock)

atomic instruction (hardware support) is needed for simple solution

## Issue #2 - Memory consistency (The two processes example)

how updates of A process becomes visible to B process

Ex on the right - can be 01 or 11

“Memory model” defines what is possible and not ⇒ different consistency

#1 - sequential consistency (most strict)

shard cache ⇒ slow on write

can jump between processes

#2 - TSO( total store order) allows “reordering of load and stores”

“store buffer” to cache writes (applied later in order) not coherent

all reads first scan the “store buffer” before L1 cache

Ex. 2 & 4 can execute directly, 1 & 3 propagate through cache

00 or 10 can happen because buffer is not applied

multi-thread is strange. single-thread behavior is maintained

#3 - barrier/fence instruction

Enforces cache flushes on fence instruction

#4 - weaker

Allows load->store, store->load, load->load, store->store reordering

Datacenter architecture

GPU

NVIDIA CUDA (computer uniform device architecture)

platforms expose synchronous execution of a massive number of threads

high-performance graphic mem (HBM2 is 3d-stacked memory)

Hardware Lottery

General-Purpose CPU Threads

❑ Moore’s Law + Dennard Scaling = Dependable performance scaling

❑ Faster general-purpose hardware available next year

o Why risk uncertain reward with specialized designs?!

❑ Resources focused on general purpose CPUs faster

Von-Neumann general-purpose CPUs

o Not very good with parallel execution

o Not much emphasis on memory bandwidth

❑ Efficient with branch-heavy expert systems

o Favors symbolic approaches to AI (LISP, Prolog)

❑ Inefficient with massively parallel matrix multiplication

o Disfavors neural networks

# GPU

CUDA

SIMD (single instruction multiple data)

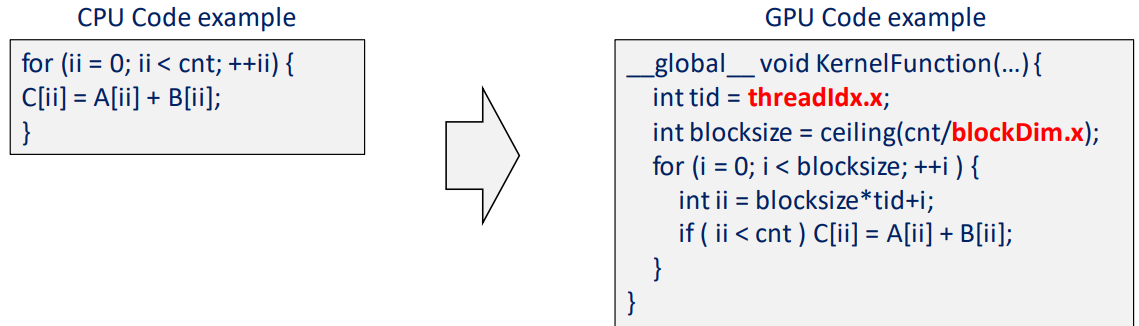
“SIMT” (Single Instruction Multiple Threads), introduced by NVIDIA

o Simply put: Identical program (“Kernel”) executed on multiple threads

o Thread ID is given as a parameter to the program,

so each thread can perform different work despite identical code

o Another kernel parameter is “block size”, the number of threads to use



Block: Multi-dimensional array of threads

o 1D, 2D, or 3D

o Threads in a block can synchronize among themselves

o Threads in a block can access **shared memory**

o CUDA (Thread, Block) ~= OpenCL (Work item, Work group)

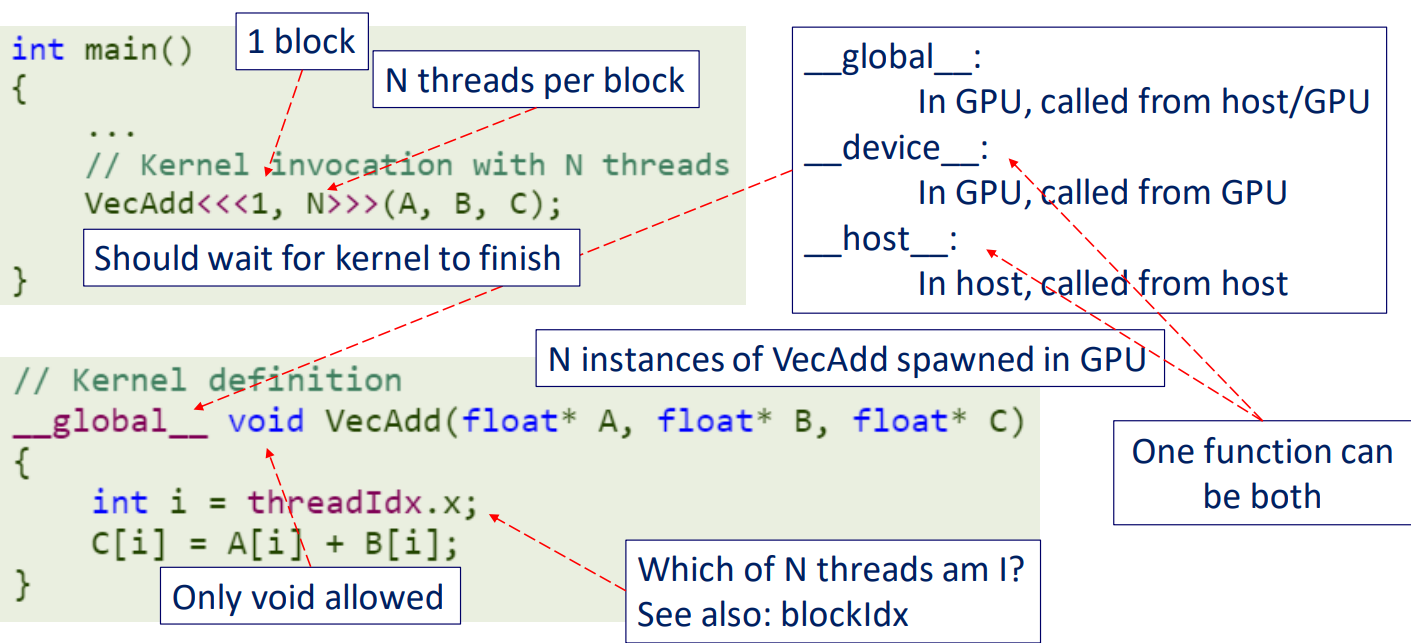
threads in a block is executed **in parallel**

Grid: Multi-dimensional array of blocks

o 1D or 2D

o Blocks in a grid can run in **parallel, or sequentially**

limited recursion of 24



Final

**Modern OS support user process isolation via “privilege levels”**

**private address space** (no other user process can access OS mem) ⇒ need VM

**schedules processes**

We can support modern OS on a machine with privileged instructions, but without VM by emulation.

PIPT is slow

VIVT needs ASID to ensure correctness.

VIPT allows parallel TLB and Cache lookup ⇒ desired

VLPT is used in L1 and PIPT is used in the rest levels.

page offset used to lookup ⇒ page offset length >= cache length

but without TLB. VIVT is usually desired in all levels. VIPT and PIPT is equally bad in L3

VM

1 KB = 2^10 bytes. 1MB = 2^20 bytes. 1 GB = 2^30 bytes

multi-level paging. Even one entry is used, the whole table is created ⇒ **DON'T forget to take unused entries into account**

Sequential Consistency (in-order inside process, arbitrary between process)

TSO consistency (more possible outputs than SC) store buffer

GPU with 1024 threads per block, where threads are organized into SIMD-warps of 32 threads. The GPU chip supports a single block. The whole chip runs at 2 GHz. For simplicity, let’s assume the off-chip memory has a capacity of 16 GB, accessed in units of 4 bytes, has infinite bandwidth, but has 100 nanosecond of latency. The GPU card is connected to the host over a 16 GB/s PCIe connection.

(a) Assuming an in-order pipeline implementation, what is the **maximum number of floating-point operations per second** this hardware can achieve, under ideal circumstances? (5 points)

**1024 threads \* 2 GHz**

(b) We are trying to perform a binary search into an array of floating-point values. If we fill the whole GPU memory with floating-point numbers (4 billion numbers), how long will it take (roughly) to finish the binary search for one query? (Please specify in nanoseconds!) (5 points)

# comparisons = log\_2 (4 billion)

time = 100 ns \* # comparisons = 100\* log\_2 ( 4 billion)

(c) In general, do you think this algorithm will be faster or slower on the GPU, compared to the CPU? Why? (5 points)

**GPU runs faster because its parallel capability dealing with large dataset**

(d) Will the answer change if the size of the array is now 32 GB? Why? (5 points)

**CPU runs faster**