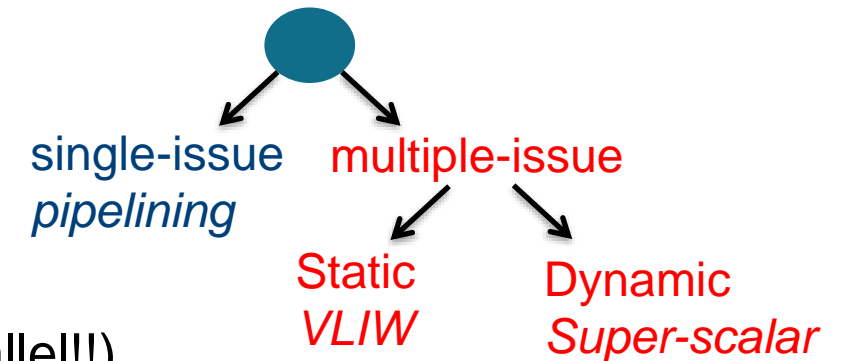


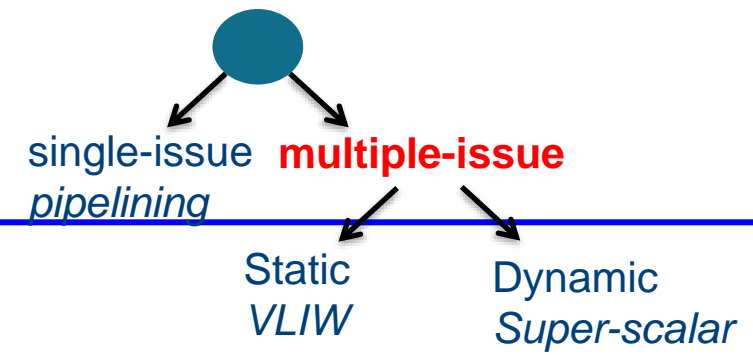
H0038a Computer architectures and the HW/SW interface

Lecture 6 Parallelism

- *Last week: Single issue functional parallelism: Pipelining*
- **Instruction level parallelism**
- Thread level parallelism
- Core level parallelism
- Challenges for parallelism?

- Pipelining = executing multiple instructions in parallel, but 1 “issue” at a time
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - More overhead, more branch penalty...
 - Multiple issue
 - Start multiple instructions per clock cycle (in parallel!!)
 - Instructions Per Cycle (IPC) $> 1 \rightarrow$ CPI < 1
 - E.g., 4GHz 4-way multiple-issue
 - 16 Giga-IPS, peak CPI = 0.25, peak IPC = 4
 - But dependencies reduce this in practice





- Static multiple issue
 - Compiler groups instructions to be issued together
 - *More deterministic, but less efficient... Why?*

- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - *Efficient, but un-deterministic... Why?*

- *Last week: Single issue functional parallelism: Pipelining*
- Instruction level parallelism
 - **Static multiple-issue (VLIW)**
 - Dynamic multiple issues (super-scalar)
- Thread level parallelism
- Core level parallelism
- Challenges for parallelism?

- VLIW processors use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations \Rightarrow Very Long Instruction Word (VLIW)
 - Issue packet or bundle – the set of instructions that are bundled together and issued in one clock cycle – think of it as one **large** instruction with multiple operations
 - The mix of instructions in the packet (bundle) is usually restricted
- Hardware modification for VLIW \rightarrow *let's see with an example*
 - Multiple parallel functional units
 - Multi-ported register files
 - Wide program bus (instruction fetch)

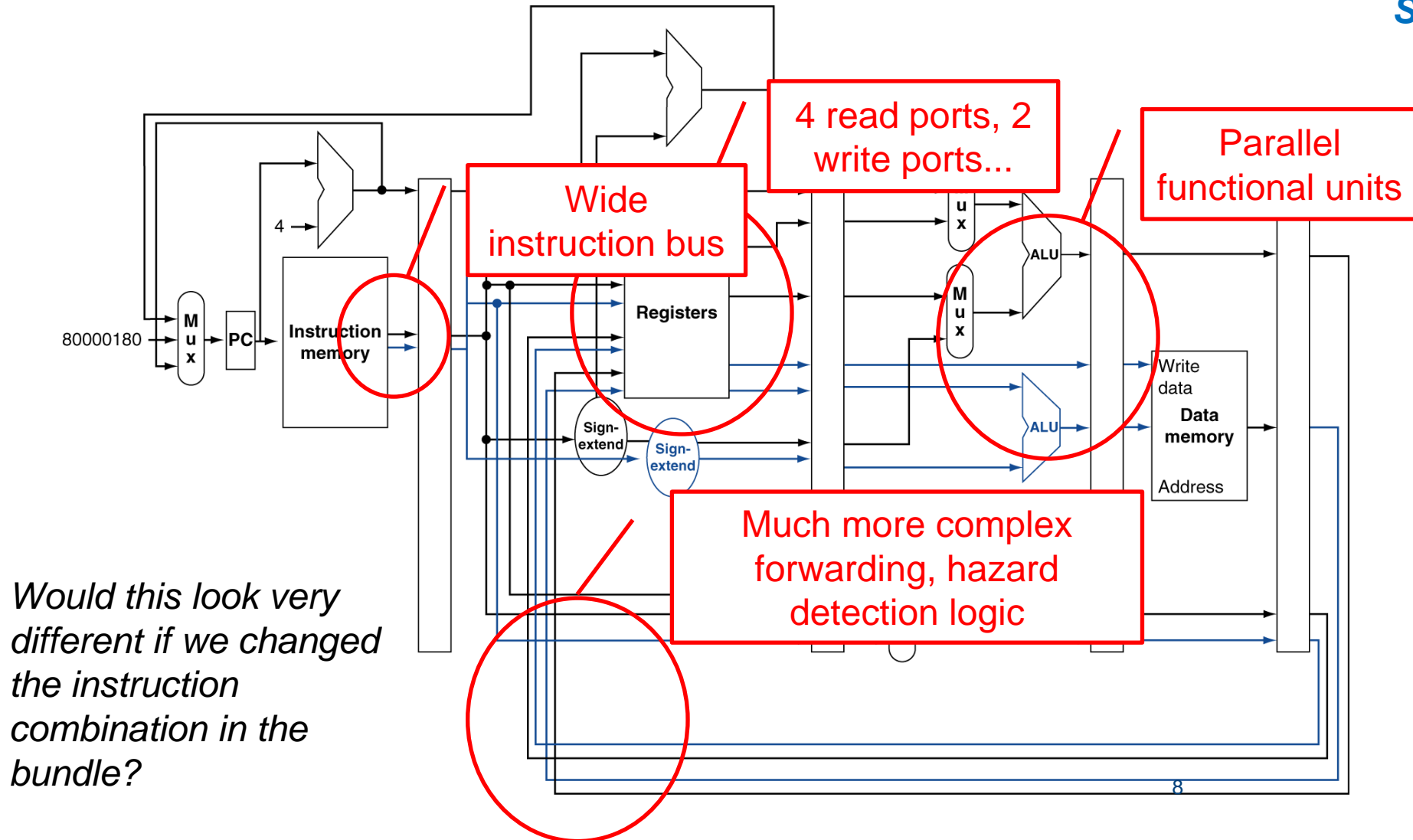
Example: Design Static Dual Issue processor

- Two-issue packets:
 - An ALU/branch instruction & a load/store instruction
 - 32 bit + 32 bit = 64-bit
 - ALU/branch, then load/store
 - Instructions are always fetched, decoded, and issued in pairs
 - If one instr of the pair can not be used, it is replaced with a nop

Address	Instruction type	Pipeline Stages						
		IF	ID	EX	MEM	WB		
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Exam: design you own X-issue processor, with specific instructions in every issue slot

See also Ripes



Would this look very different if we changed the instruction combination in the bundle?

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies within a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop's if necessary

Bigger penalties...

- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add x10, x0, x1
 - ld x2, 0(x10)
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
- *Let's look at an example!*

Example: Schedule code on previous processor

- Consider the following loop code

```
lp:   lw      $t0, 0($s1)    # $t0=array element
      add     $    , $t0, $s2 # add $s2 to $t0
      sw      $    , 0($s1)  # store result
      addi    $s1, $s1, -4    # decrement pointer
      bne     $s1, $0, lp     # branch if $s1 != 0
```

Must “schedule” the instructions to avoid pipeline stalls

Instructions in one bundle *must* be independent

Must separate use instructions from their loads by one cycle

Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies

Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:	<code>nop</code>	<code>lw \$t0, 0(\$s1)</code>	1
	<code>addi \$s1, \$s1, -4</code>	<code>nop</code>	2
	<code>add \$t0, \$t0, \$s2</code>	<code>nop</code>	3
	<code>bne \$s1, \$0, lp</code>	<code>sw \$t0, 4(\$s1)</code>	4
			5

- Four clock cycles to execute 5 instructions for a
 - CPI of 0.8 (versus the best case of 0.5)
 - IPC of 1.25 (versus the best case of 2.0)
 - NOP's don't count towards performance !!

- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead

- Use different registers per replication
 - Called “**register renaming**”
 - To avoid over-writing, or false dependencies
 - **Compiler** does this (offline)!

Loop Unrolling Example (4 times)

	ALU/branch	Load/store	cycle
Loop:	addi \$s1 , \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	add \$t0 , \$t0, \$s2	lw \$t2, 8(\$s1)	3
	add \$t1 , \$t1, \$s2	lw \$t3, 4(\$s1)	4
	add \$t2 , \$t2, \$s2	sw \$t0 , 16(\$s1)	5
	add \$t3 , \$t4, \$s2	sw \$t1 , 12(\$s1)	6
	nop	sw \$t2 , 8(\$s1)	7
	bne \$s1 , \$zero, Loop	sw \$t3 , 4(\$s1)	8

- $IPC = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

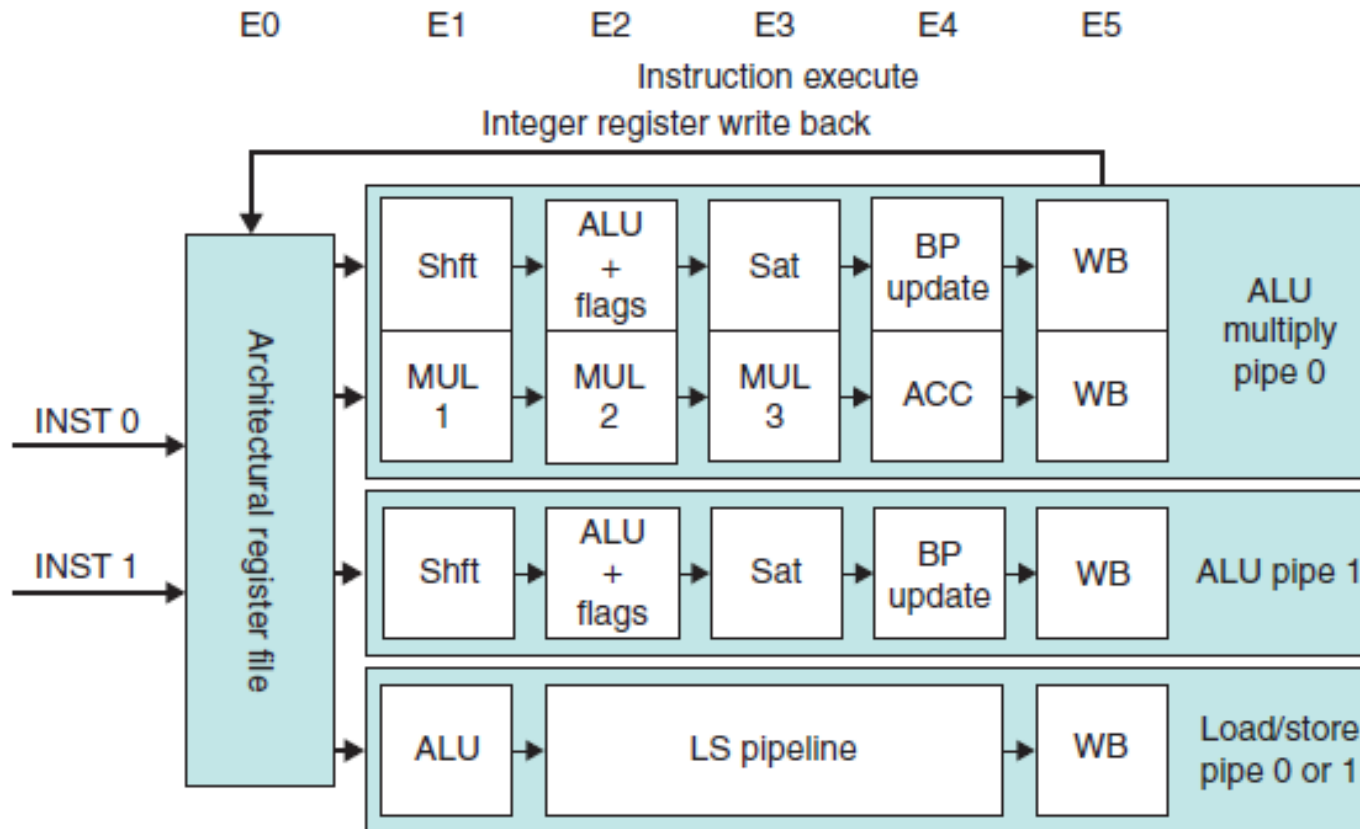
Summary:

Compiler Support for VLIW Processors

- The compiler packs groups of **independent** instructions into the bundle
 - Done by code re-ordering
- The compiler uses loop unrolling to expose more ILP
 - = increasing basic block size!
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- VLIW's mostly depend on the compiler for branch handling
 - Loop unrolling reduces the number of conditional branches
 - Code re-ordering
 - (Penalty of misprediction would be very large...)

Example: ARM Cortex-A8

- Present in (older) Apple processor in the iPad, as well as the processor in the Motorola Droid and the iPhones
- Two issue statically scheduled



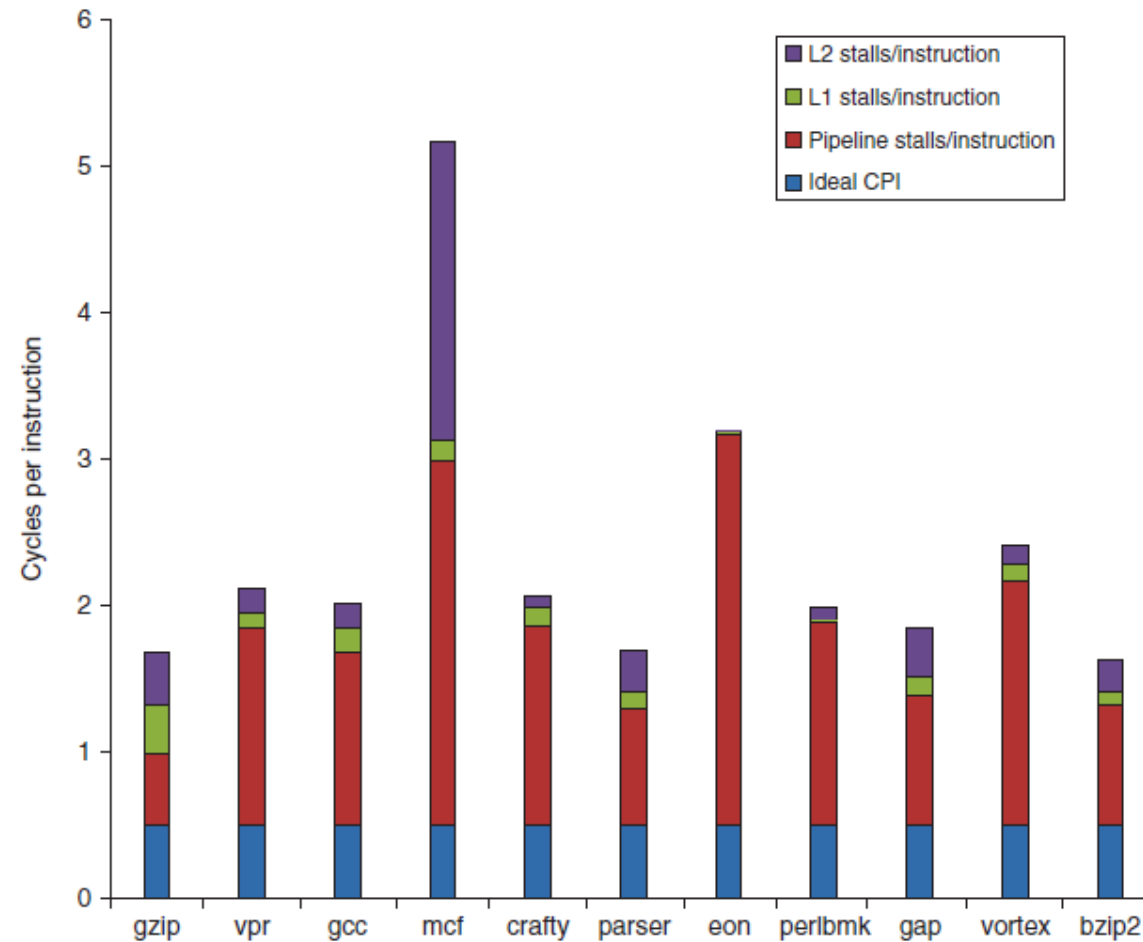
Hennessey & Patterson

(BP = branch predictor)

16

Example: ARM Cortex-A8

- Two issue statically scheduled: $CPI=0.5$?



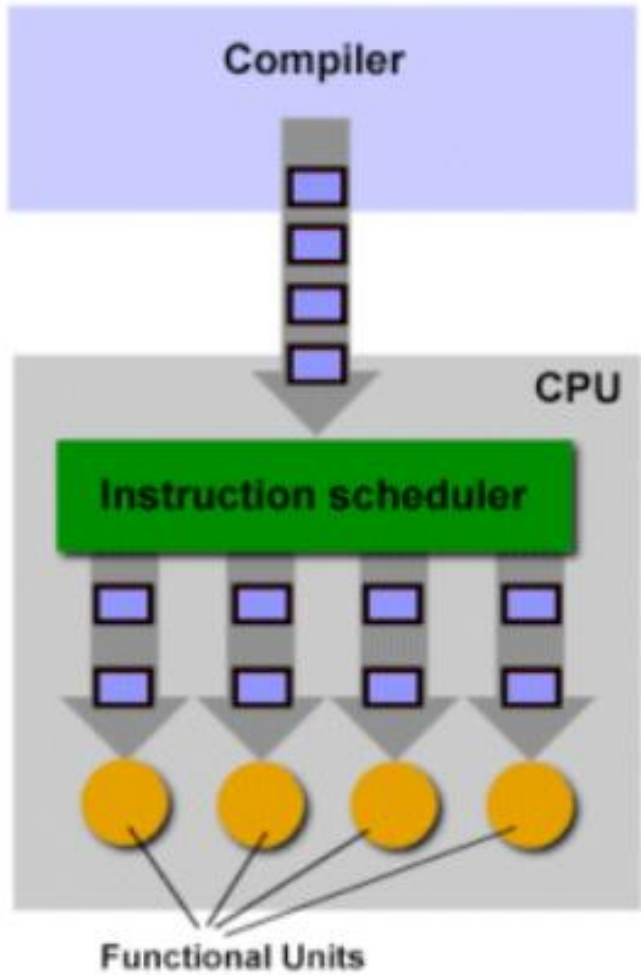
*Can not get rid of all inefficiencies
 → Better scheduling dynamically?*

Hennessey &
Patterson

(BP = branch
predictor)

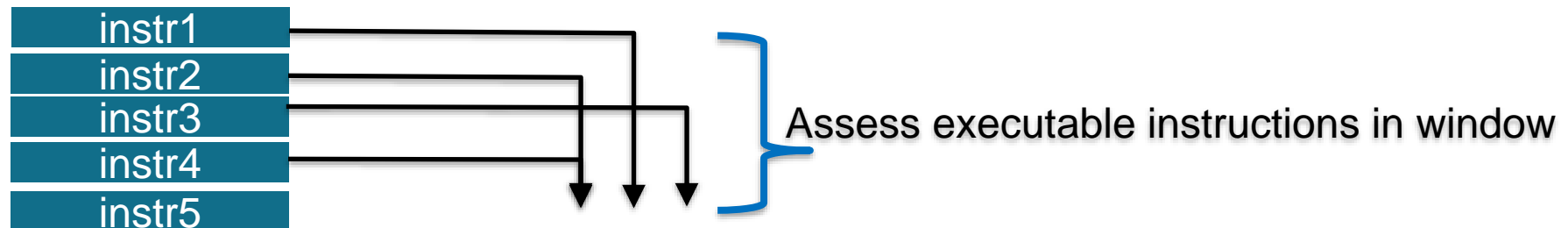
17

- *Last week: Single issue functional parallelism: Pipelining*
- **Instruction level parallelism**
 - Static multiple-issue (VLIW)
 - **Dynamic multiple issues (super-scalar)**
- Thread level parallelism
- Core level parallelism
- Challenges for parallelism?



**Dynamic Superscalar
Instruction Scheduling**

- “Superscalar” processors
 - CPU dynamically decides with dedicated hardware whether to issue 0, 1, 2, ... instructions each cycle
 - Avoiding structural and data hazards
 - Avoids the need for compiler scheduling
 - Though it may still help
 - Code guaranteed by HW to execute correctly
- ➔ Portable across platforms

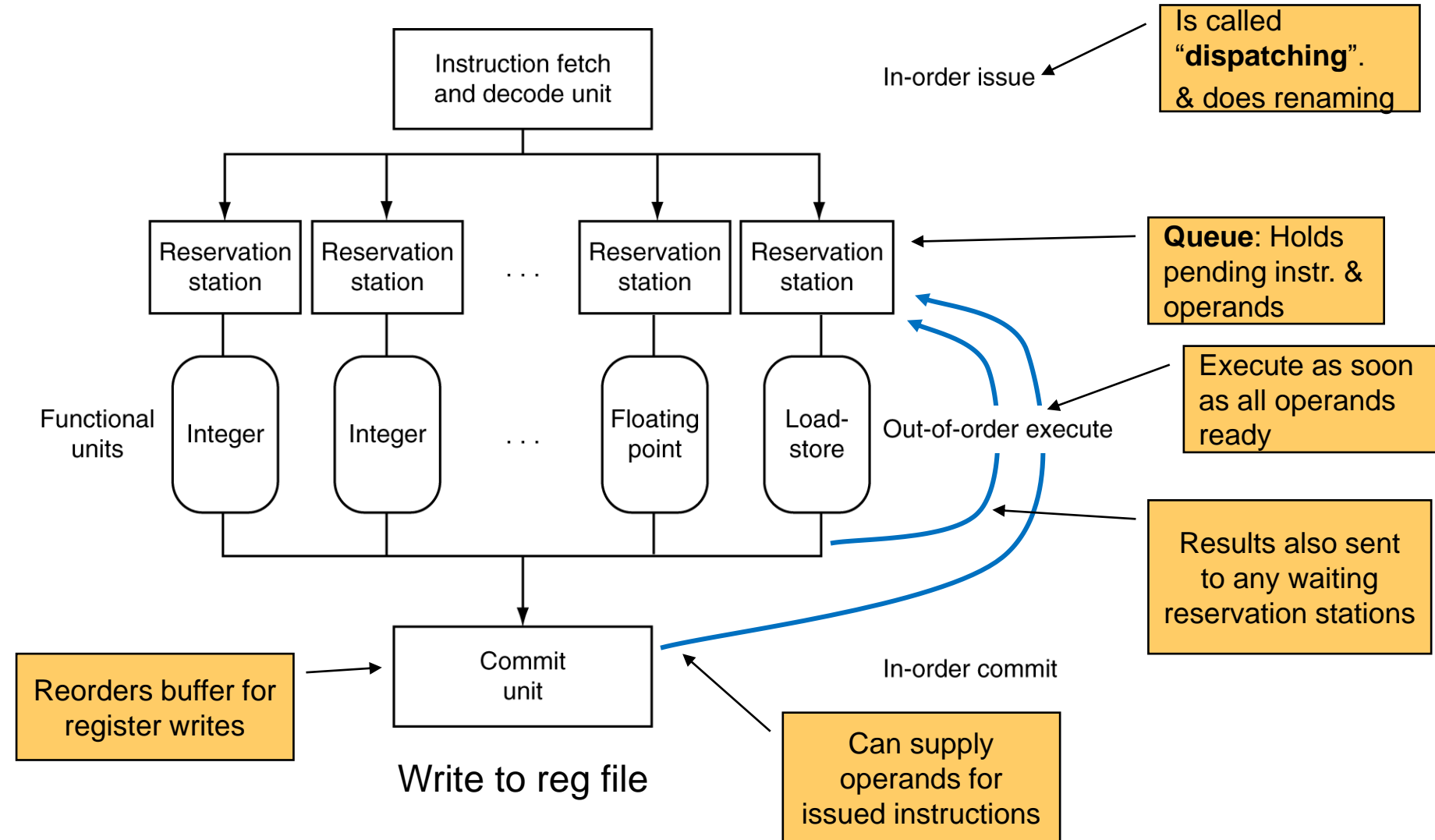


- Example

```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x1, x23, x3
sd    x5, 20(x23)
```

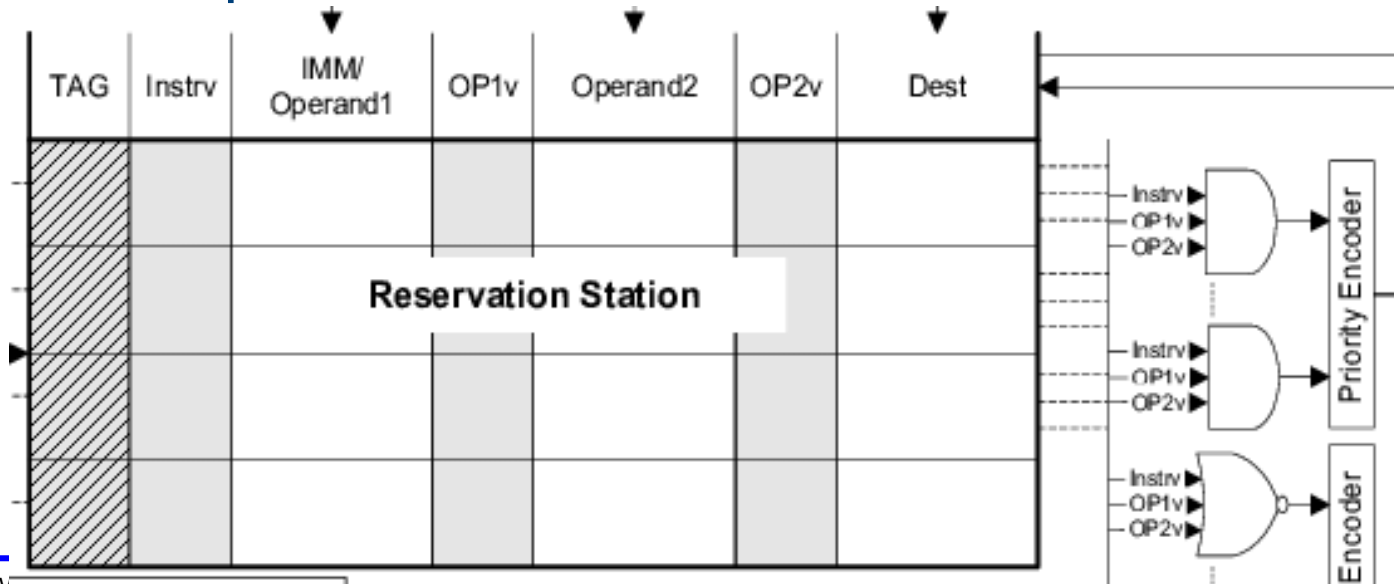
- Can start sub while add is waiting for ld
-
- CPU issue instructions that are ready for execution
 - Allows the CPU to execute instructions **out-of-order** to avoid stalls

Dynamically Scheduled CPU



Reservation stations

- Reservation stations keep list of instructions that are in the queue to be executed on a specific functional unit.
- Each instruction stored with either
 - Operand value (e.g. for x2 in the example of 2 slides back)
 - Tag to operand source station for results not ready yet (e.g. for x31 in the example of 2 slides back → (ld))
- Instruction with all operands valid can fire to functional unit



■ Example

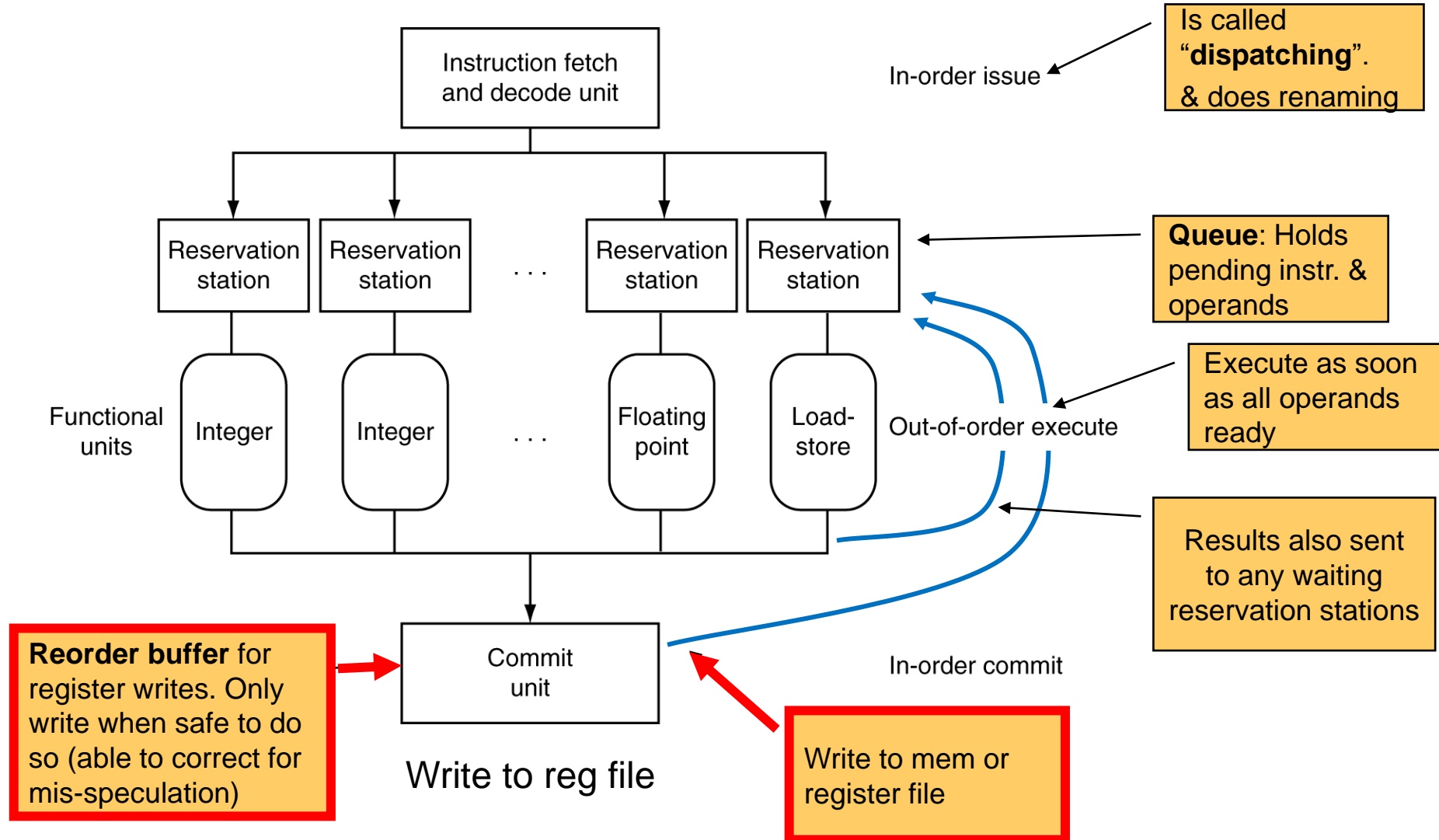
```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x1, x23, x3
sd    x5, 20(x23)
```

- Can start sub while add is waiting for ld

■ CPU issue instructions that are ready for execution

- Allows the CPU to **execute** instructions **out-of-order** to avoid stalls
- But important that results are **committed in-order**
 - Otherwise output would depend on instruction order!

Dynamically Scheduled CPU



- Maintains an ordered list of the instructions

		Clock Cycle		
		X	X+1	X+2
1	ld dest: x31	WB		
2	add dest: x1	--	WB	
3	sub dest: x1	W	W	WB
4	sd dest: Mem(20(x23))	--	--	

- Puts the instruction results back in the original program order after the instructions have finished execution
- Uses a circular buffer (buffer space ~ number of “in-flight” instructions)

- Superscalar increases data dependencies and branch penalties.
- Speculation (=“guessing”) is used to allow issuing and execution of future instructions that (may) depend on the speculated instruction
 - Branch speculation: Predict branch and continue issuing
 - Don’t commit until branch outcome determined
 - Roll back in case of wrong prediction (see earlier)
 - Load speculation: Predict loaded value
 - Load can take a lot of cycles (cache misses,...)
 - Avoid load and cache miss delay
 - Predict the effective address (*base address of ld instruction might still change*)
 - Load before completing outstanding stores (*memory content of ld target might still change*)
 - Don’t commit load until speculation cleared

Commit consequences of speculation?

- Speculative instruction (e.g. instruction after branch prediction)
 - Start operation like normal instruction
 - Propagate result up to commit buffer
 - Before commit: check whether guess was right
 - If so, complete the commit
 - If not, roll-back and do the right thing, flush the rest of the commit buffer
- Results held until it determines they are actually correct (in commit unit), otherwise flush!

- **Static**: Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess→ *VLIW*

- **Dynamic**: Hardware can look ahead for instr. to execute
 - Buffer results until it determines they are actually correct (in commit unit)
 - e.g. branch outcome correct., or other type of speculation cleared
 - Flush buffers on incorrect speculation→ *Superscalar*

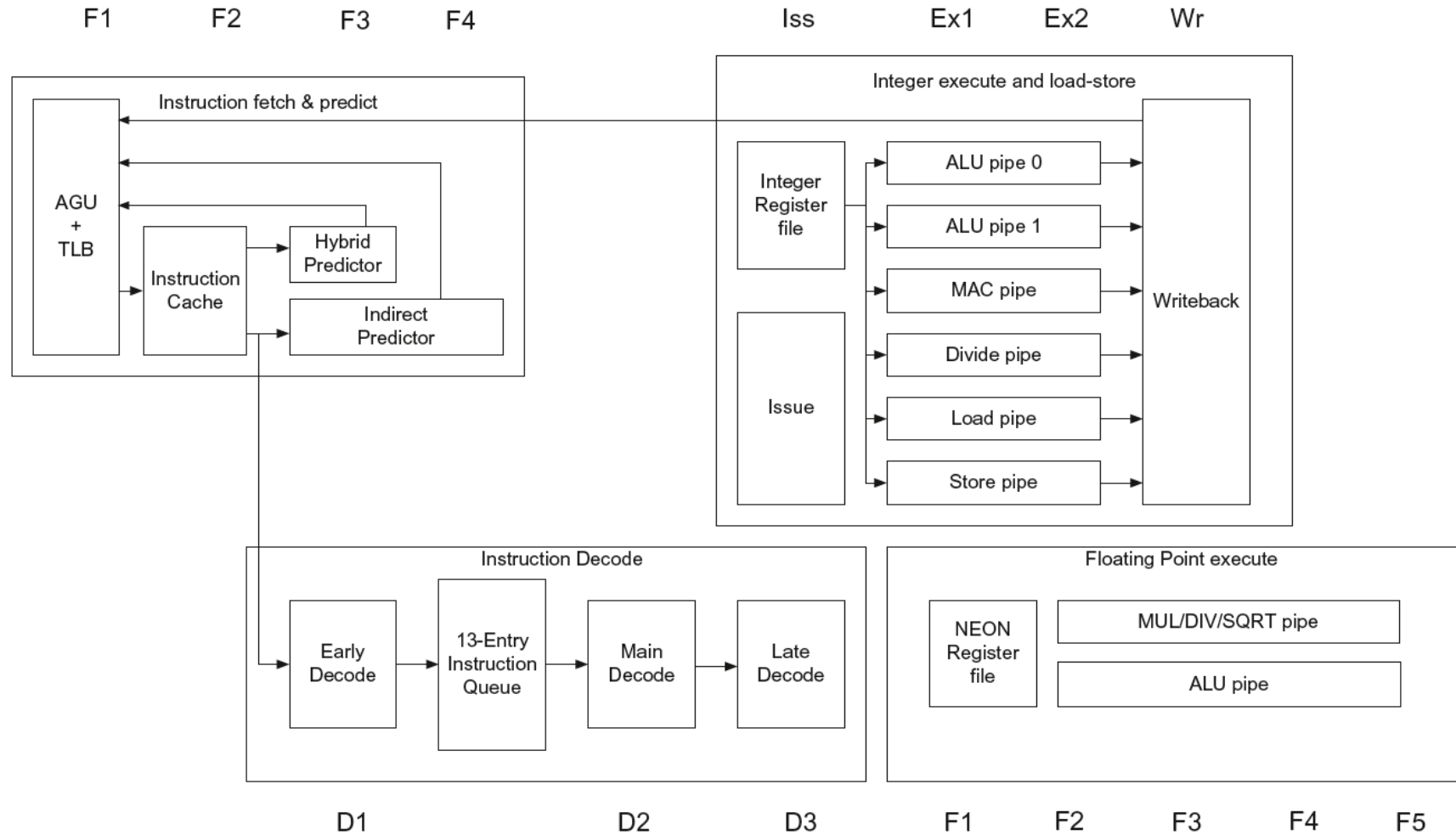
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
 - e.g. cache misses
- Can't always schedule around branches
 - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
- Again: Performance at the expense of determinism...

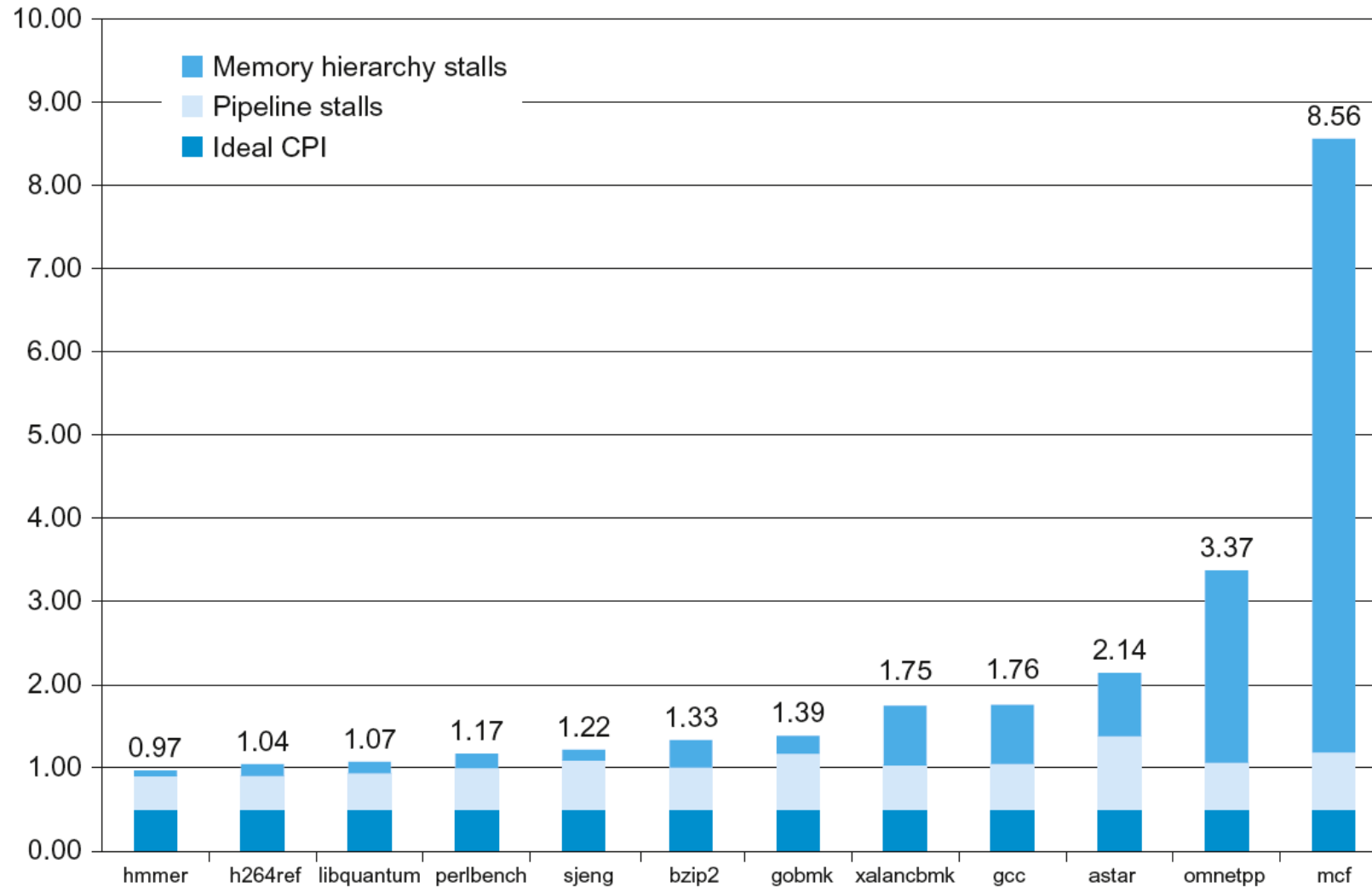
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1 st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2 nd level caches/core	128-2048 KiB	256 KiB (per core)
3 rd level caches (shared)	(platform dependent)	2-8 MB

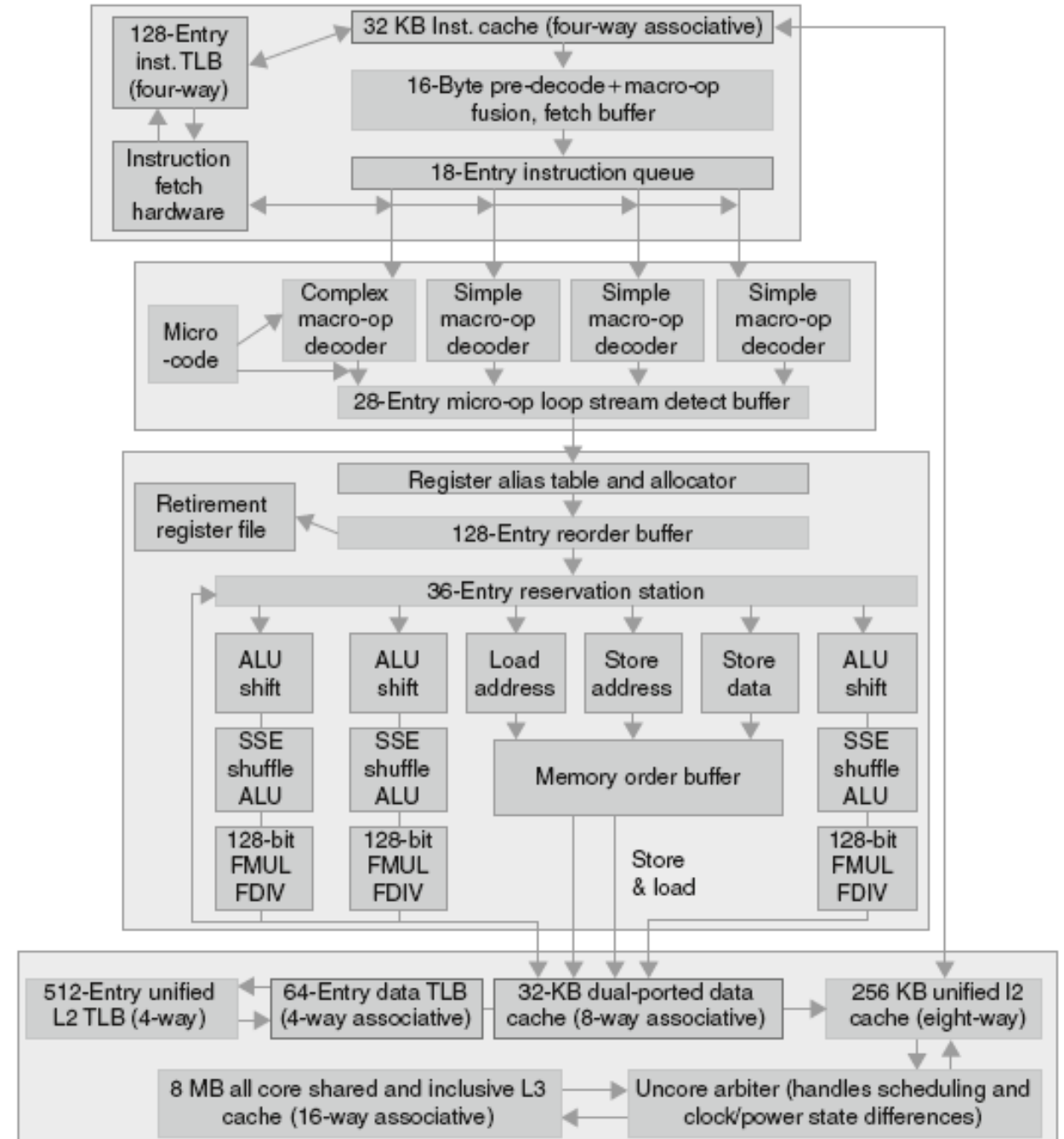
ARM Cortex-A53 Pipeline

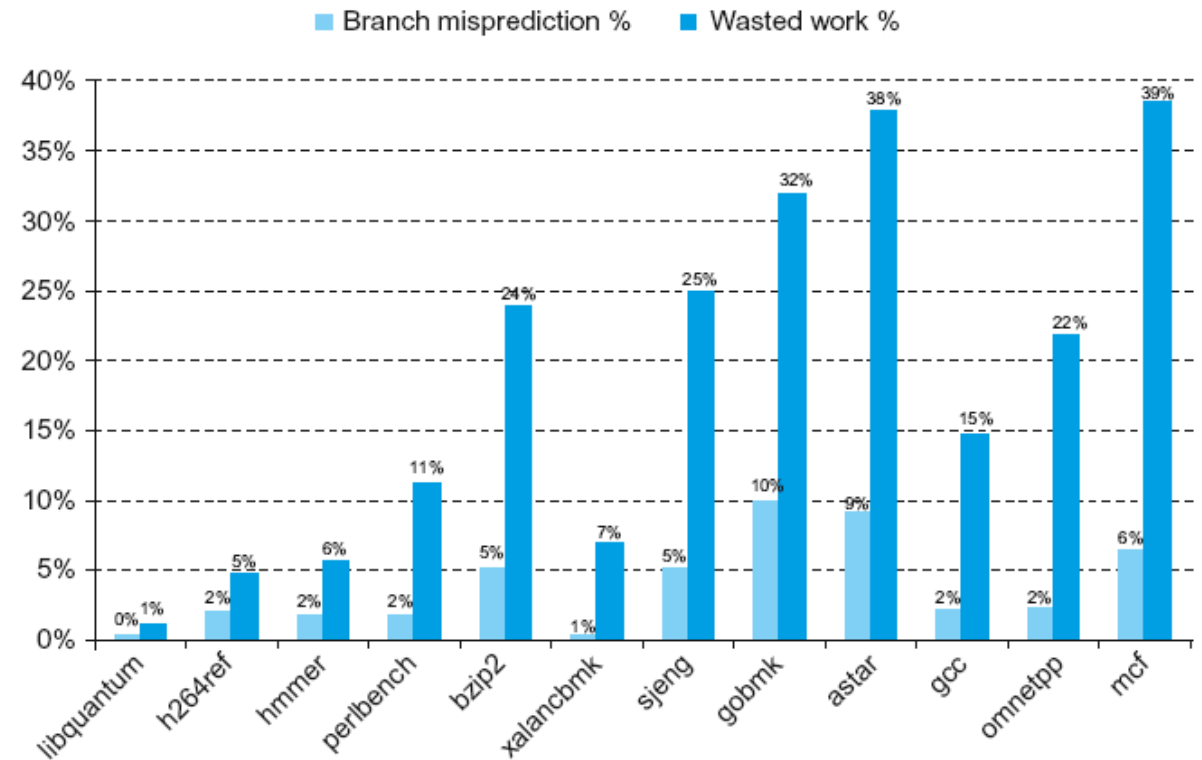
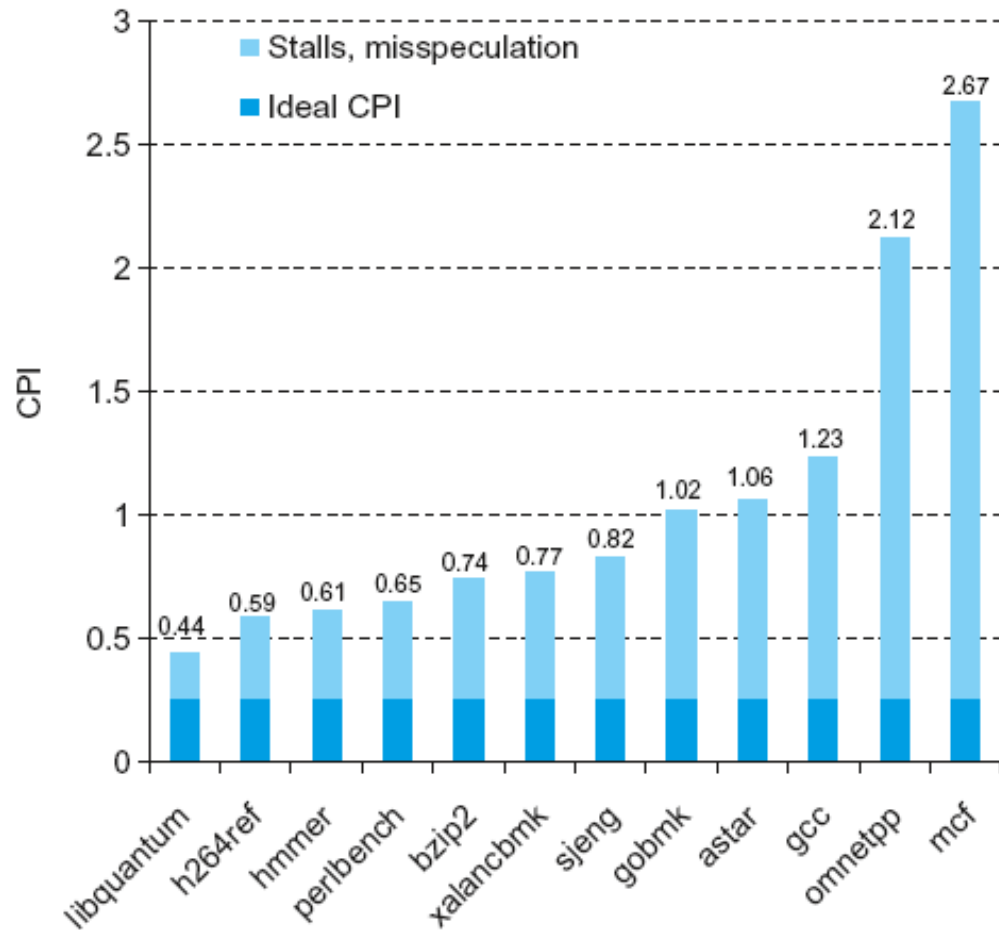


ARM Cortex-A53 Performance



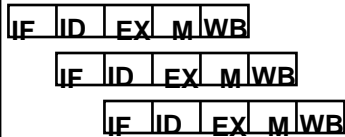
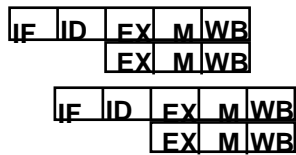
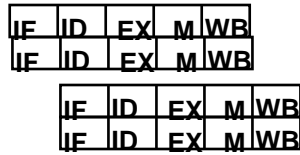
Do you recognize the different super-scalar processor blocks?





Strongly benchmark dependent...

Pipelined vs SS vs VLIW

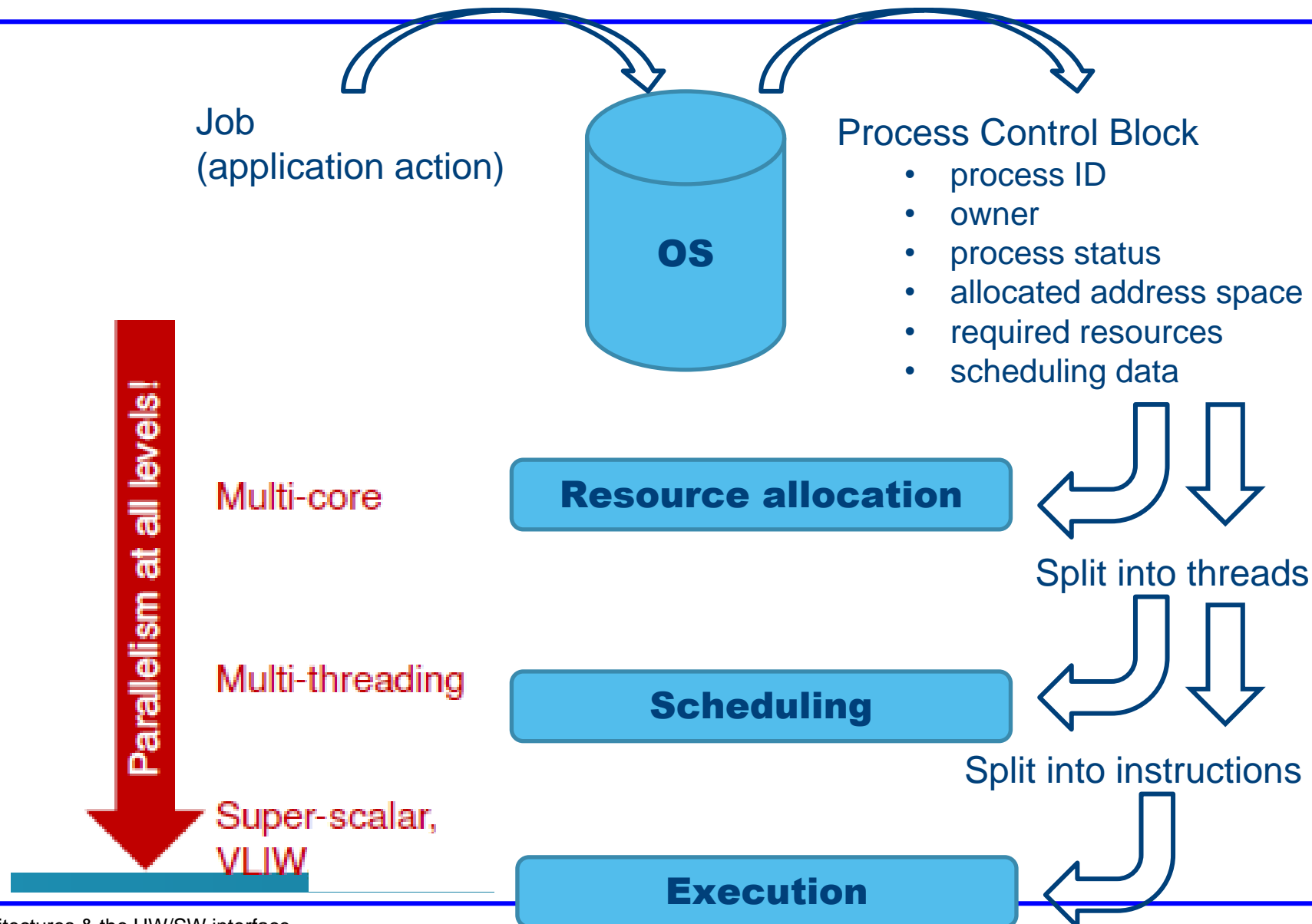
	Pipelined	VLIW	Superscalar
Instr size	fixed size	fixed size (but large)	fixed size
Key Issues	data forwarding, hazards	(compiler) code scheduling	hardware dependency resolution
Instruction flow			

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
- Some parallelism is hard to expose
 - Limited window size during instruction issue
- Memory delays and limited bandwidth
 - Hard to keep pipelines full
 - Speculation can help if done well
 - Still many underutilized functional units → parallelism at other levels

- *Last week: Single issue functional parallelism: Pipelining*
- Instruction level parallelism
 - Static multiple-issue (VLIW)
 - Dynamic multiple issues (super-scalar)
- **Thread level parallelism**
- Core level parallelism
- Challenges for parallelism?

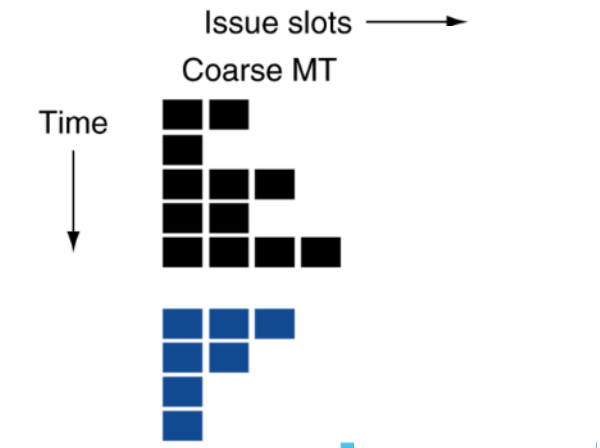
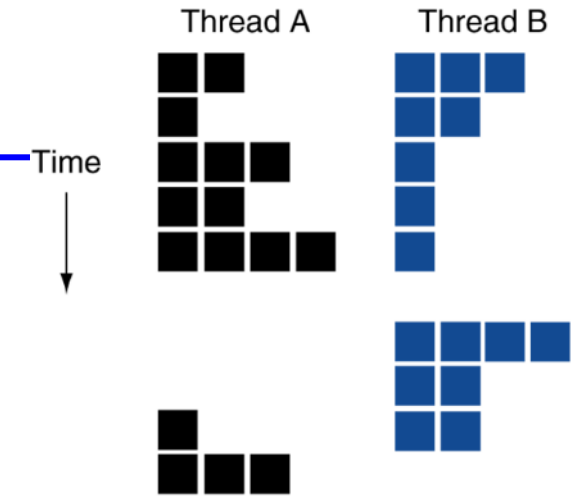
Job – process – thread



- **Program:** An executable file residing on the disk (=passive)
 - **Process:** Executing (sub)instance of a program (=active)
 - **Thread:** Unit of execution that can be autonomously scheduled (breaks into instructions)
-
- Each process has one or more threads
 - Each thread belong to one process

- More than 1 thread can be dispatched to 1 core

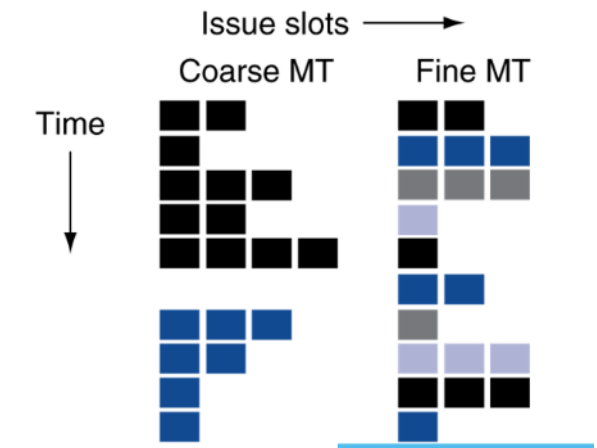
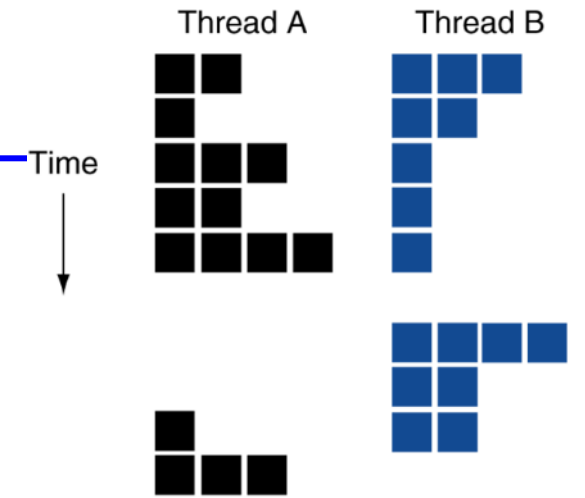
- Coarse-grain multithreading
 - Switch on long stall of treat (e.g., L2-cache miss)
 - Minor hardware impact, but doesn't hide short stalls (eg, data hazards)



- Performing multiple threads in parallel

- Fine-grain multithreading

- Switch threads after each cycle
- Interleave instruction execution
- If one thread stalls, others are executed

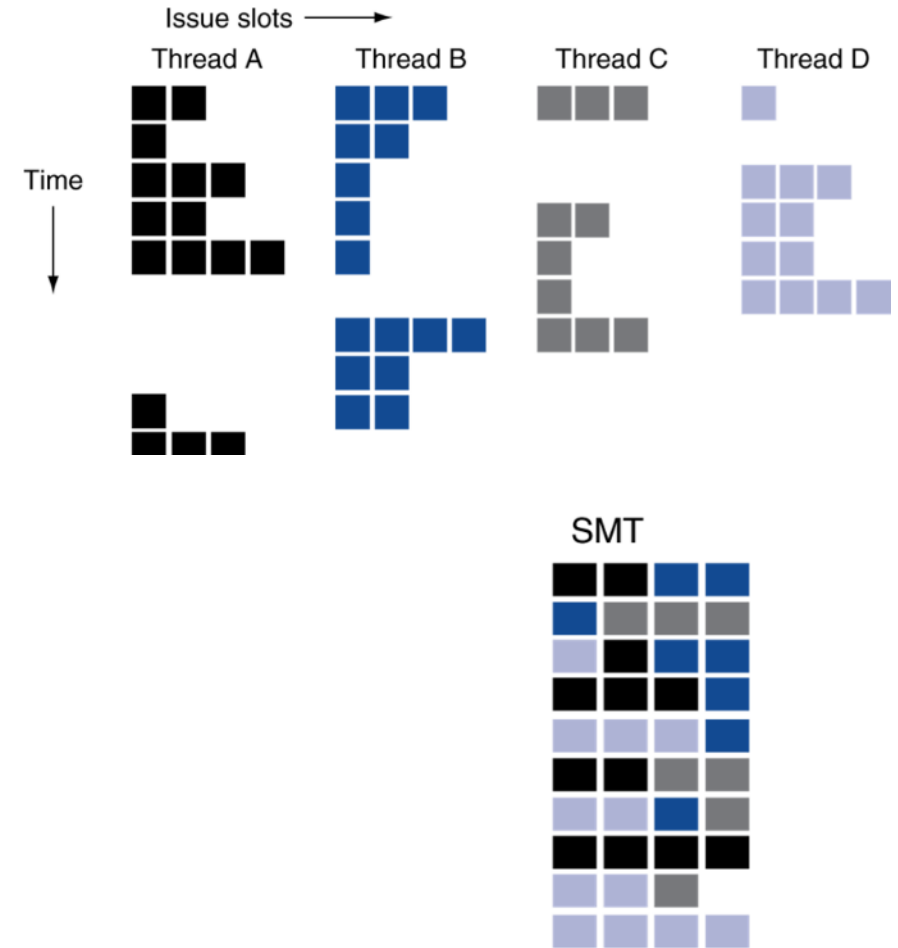


- In multiple-issue dynamically scheduled processor
 - Mix instructions from multiple threads
 - Instructions from independent threads execute when function units are available
 - Within threads, dependencies handled by scheduling and register renaming

- Limit amount of concurrent threads

(duplicated registers, shared function units and caches, humongous bookkeeping in reservation stations, commit buffer,....)

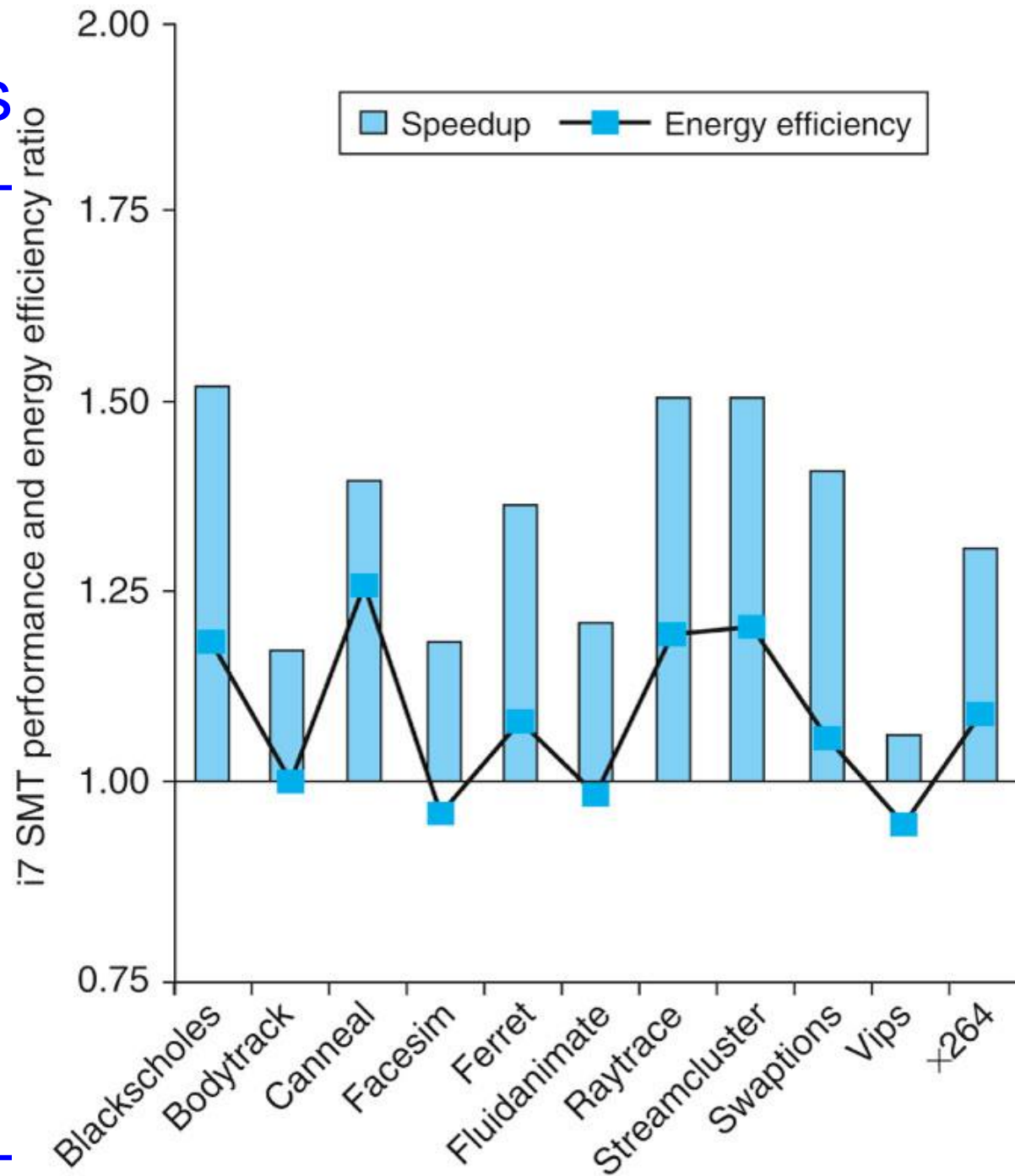
Example: 2 threads in Intel Pentium-4 HT



SMT benefits

On an Intel i7 processor (2 threads)
for the PARSEC benchmarks →

(i5's are i7's with SMT disabled
→ binning!)

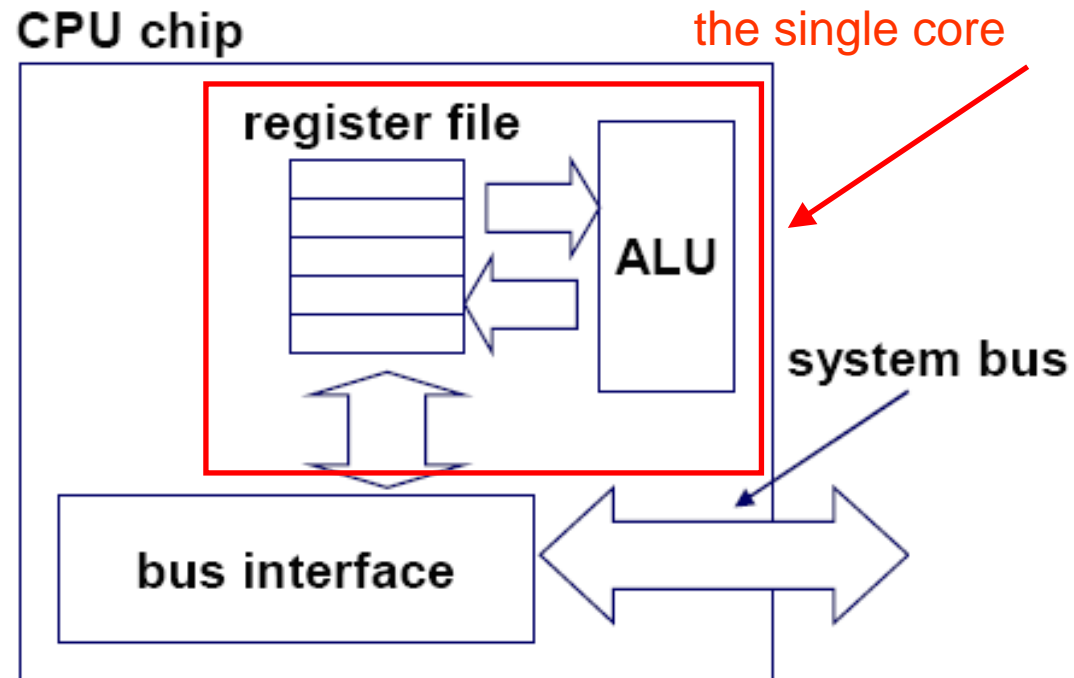


- Will it survive? In what form?
- Power considerations \Rightarrow simplified microarchitectures
 - Simpler forms of multithreading
- Tolerating cache-miss latency
 - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

- *Last week: Single issue functional parallelism: Pipelining*
- Instruction level parallelism
 - Static multiple-issue (VLIW)
 - Dynamic multiple issues (super-scalar)
- Thread level parallelism
- **Core level parallelism**
- Challenges for parallelism?

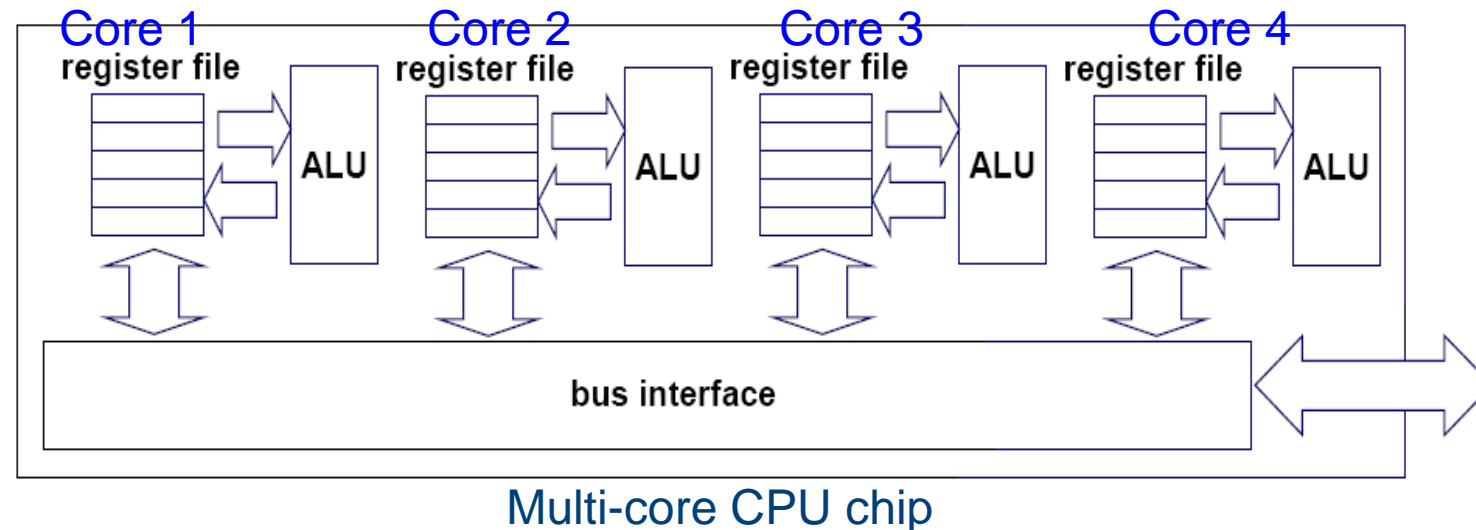
Motivation multi-processors

- Many new applications are multithreaded
- Single-core superscalar processors cannot fully exploit MT
 - Due to memory bandwidth and HW overhead
 - Due to power density!

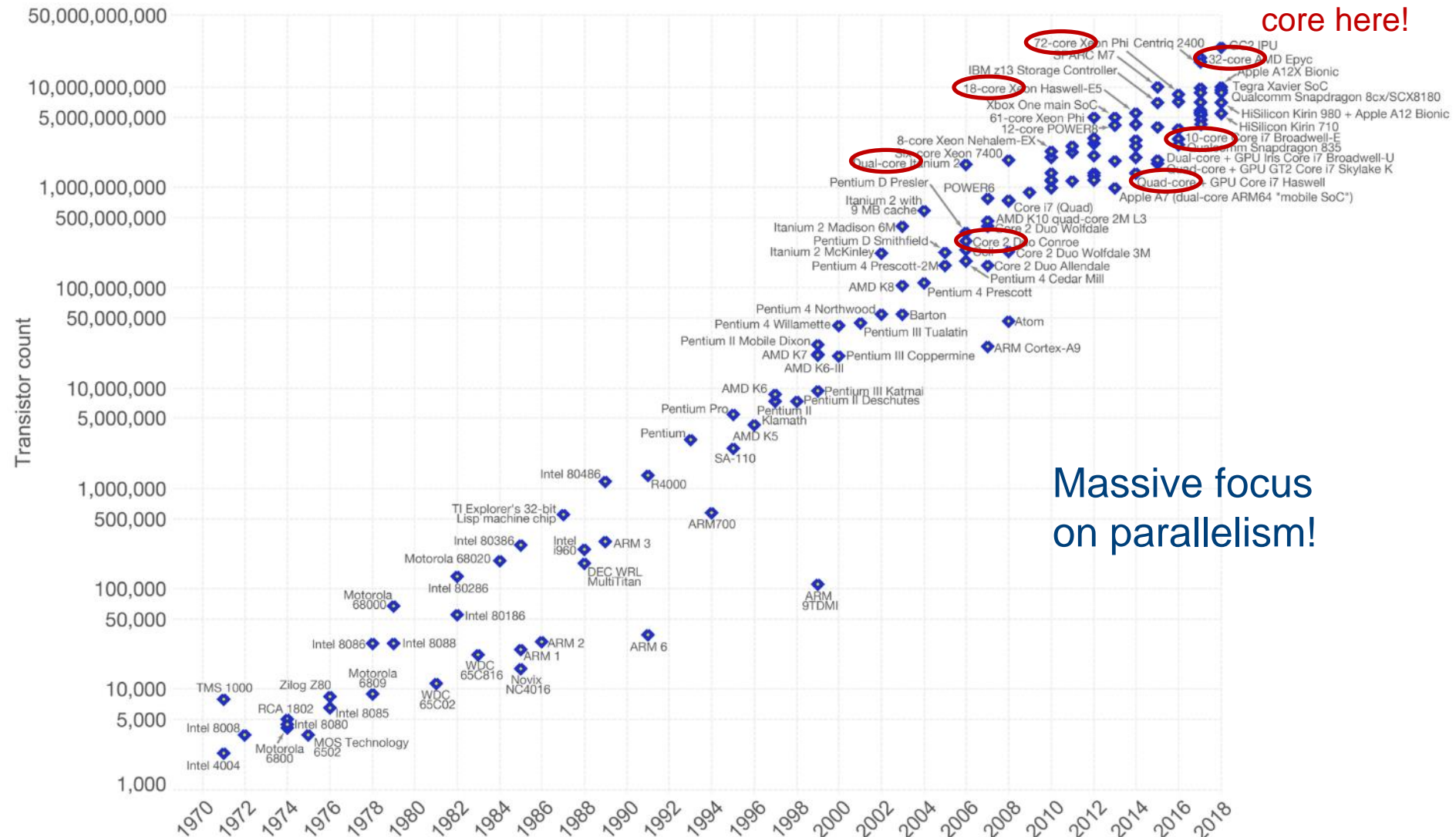


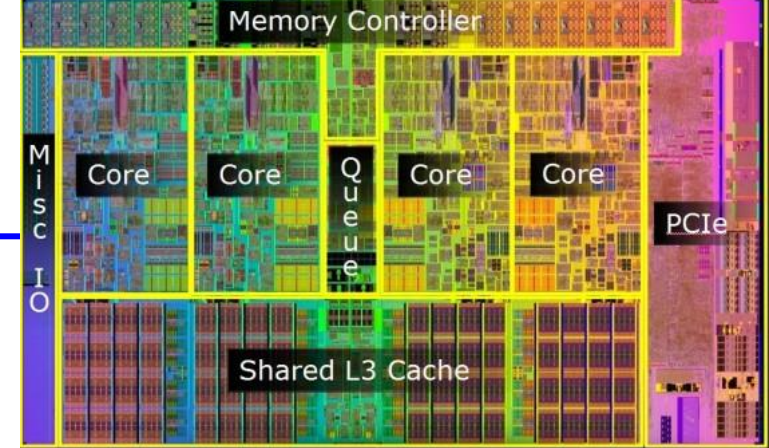
Slides based on slides Jernej Barbic

- Goal:
 - Connecting multiple cores to get higher performance at constant Watt/cm²
 - Scalability, availability
- Job-level (process-level) parallelism
 - High throughput for independent jobs



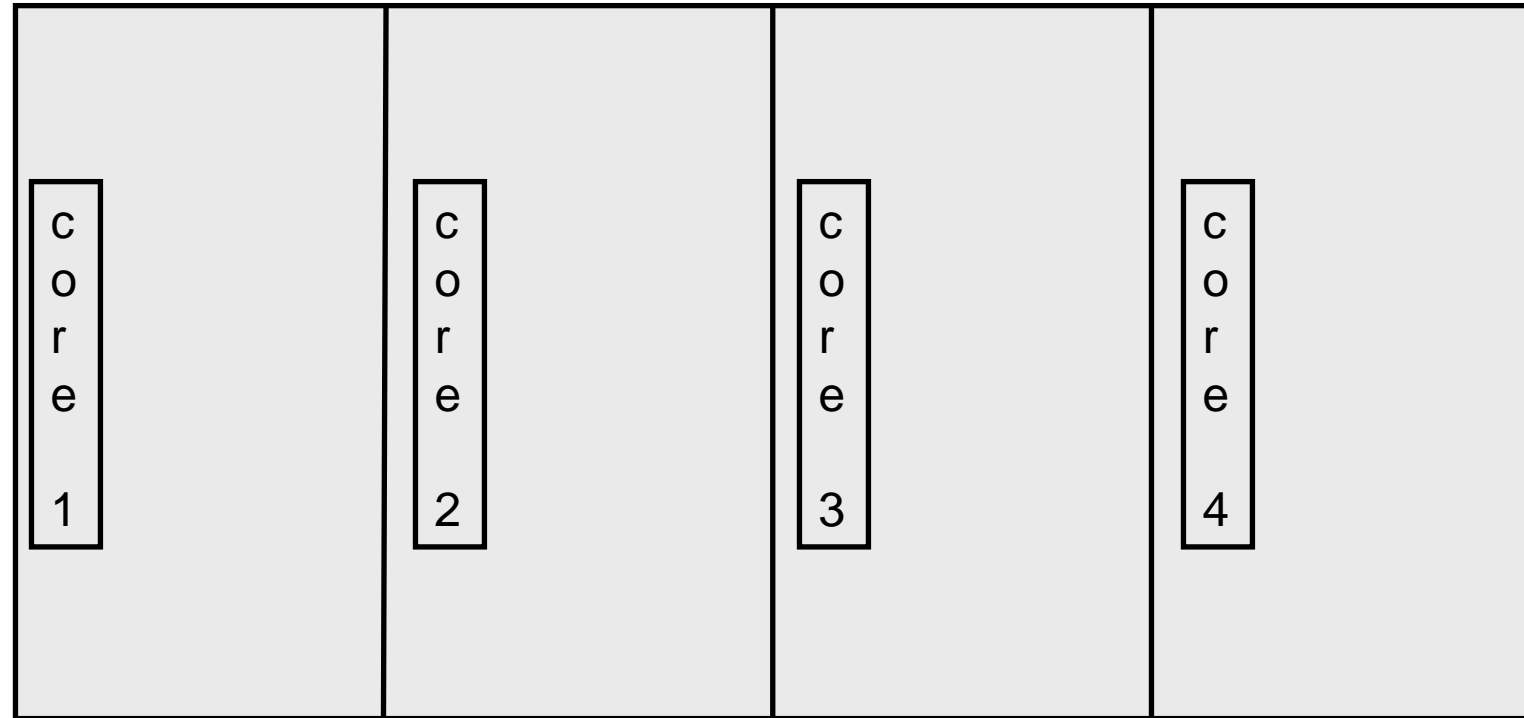
Massive focus on parallelism!





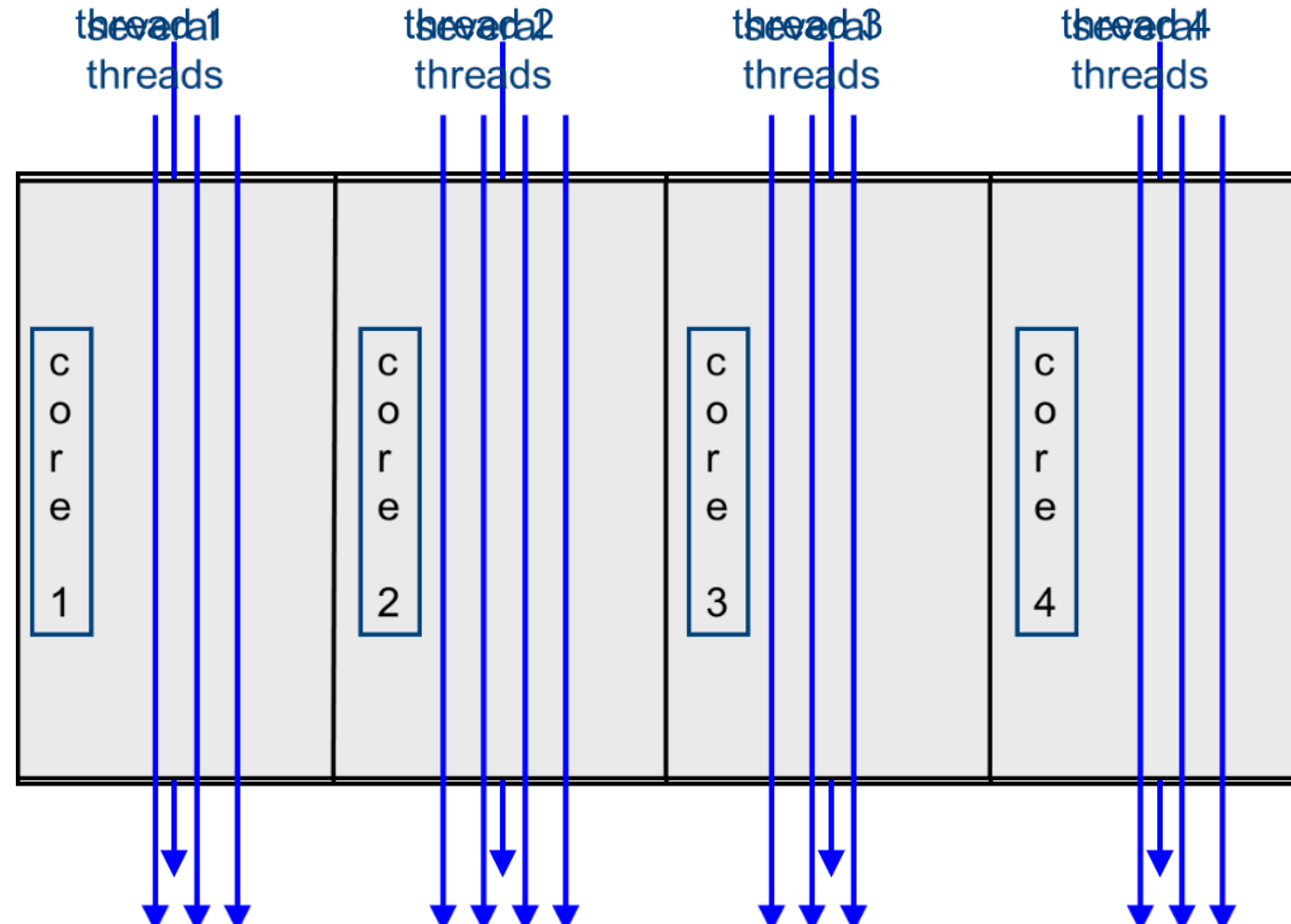
■ Multi-core microprocessors

- Multi-processor with multiple processors (cores) on same die



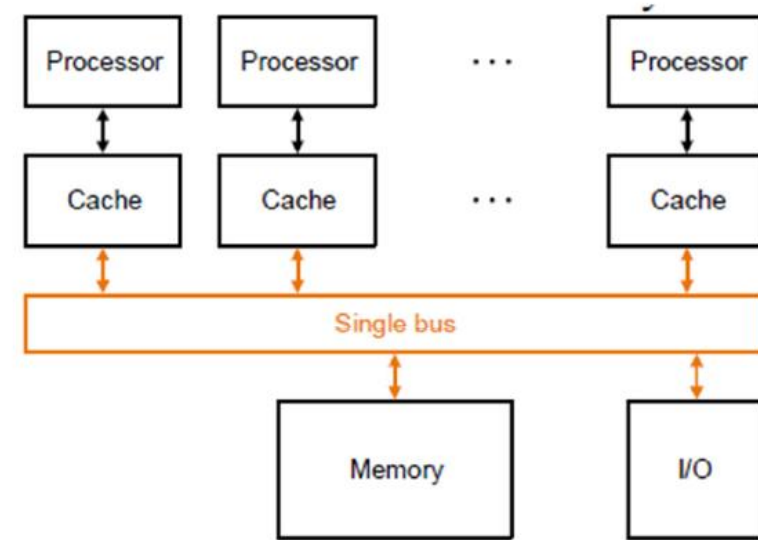
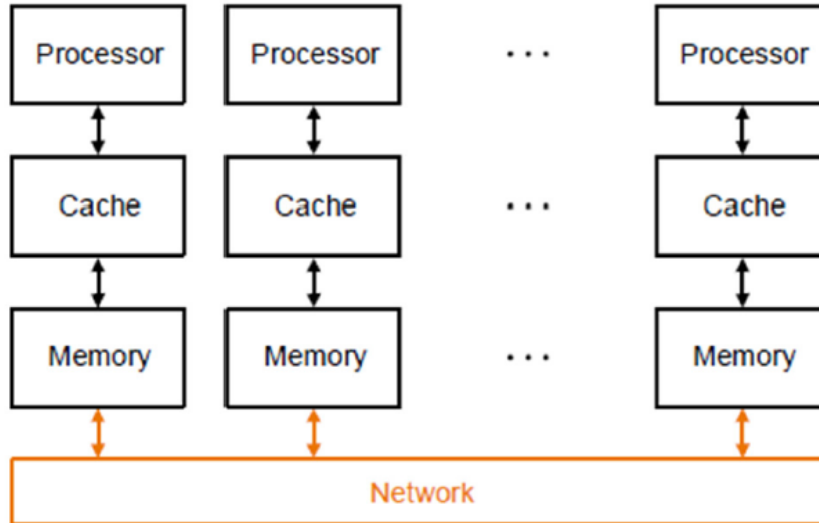
The cores run in parallel

Parallel processing program: Single program run on multiple cores

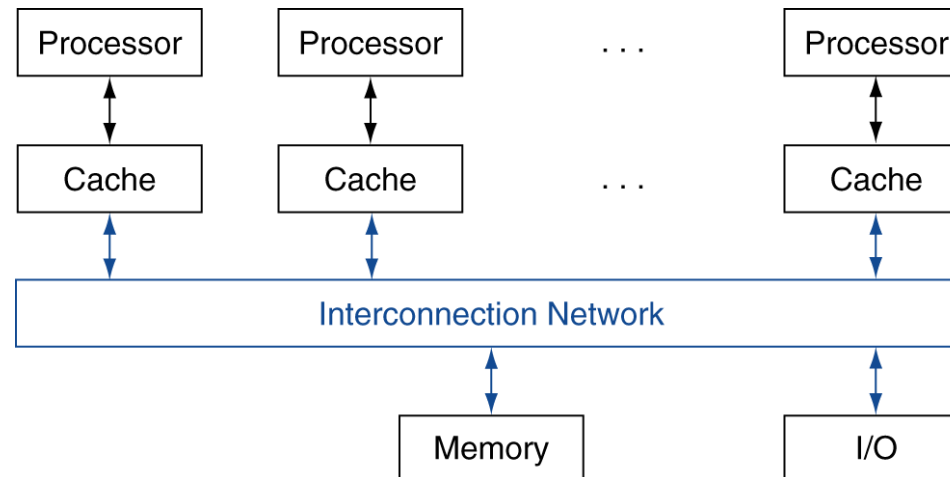


Implementation implications: memory and interconnect

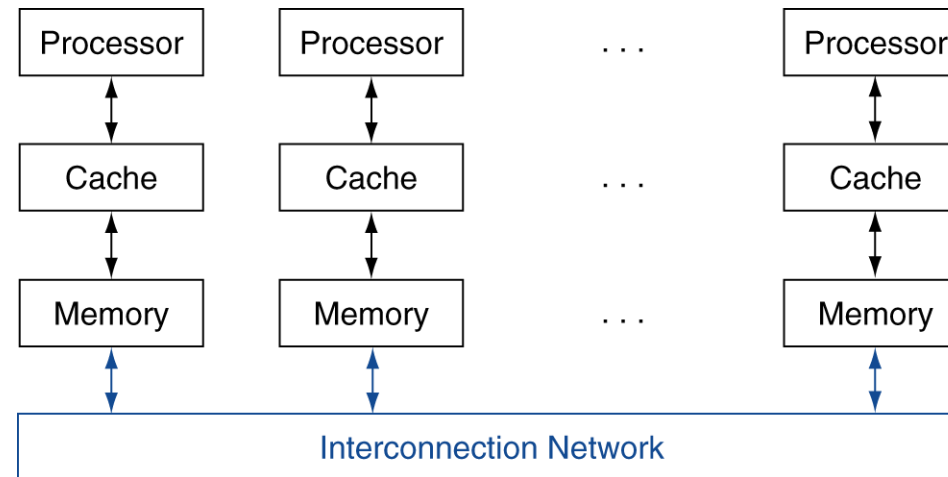
- Different cores have to exchange messages and data
- This can be done at different levels and is reflected in the memory hierarchy
 1. Distributed memory architecture with message passing
 2. Shared memory architecture: SMP: Symmetric multiprocessors



- SMP: 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)



- Memory distributed among processors, with private physical address space each
- Explicitly send messages between processors, instructed by SW designers... Very difficult to program!
- Best for applications where little communication between jobs
- E.g. IBM's Cell processor in PS3



- *Last week: Single issue functional parallelism: Pipelining*
- Instruction level parallelism
 - Static multiple-issue (VLIW)
 - Dynamic multiple issues (super-scalar)
- Thread level parallelism
- Core level parallelism
- **Challenges for parallelism?**
 - **Amdahl's law**
 - Arithmetic intensity (memory bandwidth)

Warning for parallelism!

- Often requires explicitly parallel programming
 - Hard to do
 - Programming for performance
 - Load balancing
 - Optimizing communication and synchronization
 - Amdahl's law!

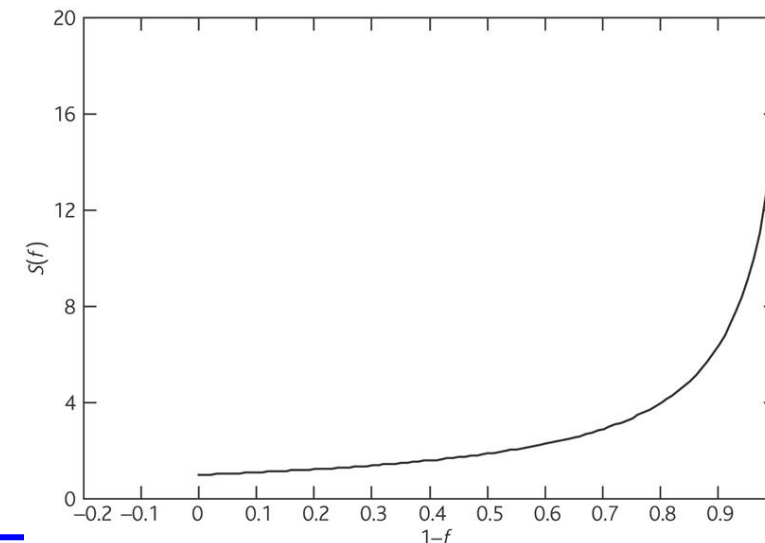
Intel fellow Shekhar Borkar stated that *“The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years.”*
→ Amdahl's law

- *Parallelism does not always help!*
- f = program fraction that *cannot* be sped up (e.g. parallelised)
 p = improvement factor (=parallelism)
- Speedup S_p

$$S_p = \frac{T_{\text{original}}}{T_{\text{improved}}} = \frac{T_{\text{original}}}{f \cdot T_{\text{original}} + (1-f) \cdot T_{\text{original}} / p} = \frac{p}{1 + (p-1) \cdot f}$$

= **Amdahl's law**

- ➔ don't speed up unless f is low
- ➔ make the common case fast



Scaling Example

- Workload: sum of 10 scalars, and 10×10 matrix sum
 - Speed up from 10 to 100 processors?
- 1 processor: Time = $(10 + 100) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Speedup = $110/20 = 5.5$ (55% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
 - Speedup = $110/11 = 10$ (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100×100 ?
- 1 processor: Time = $(10 + 10000) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - Speedup = $10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - Speedup = $10010/110 = 91$ (91% of potential)
- Assuming load balanced

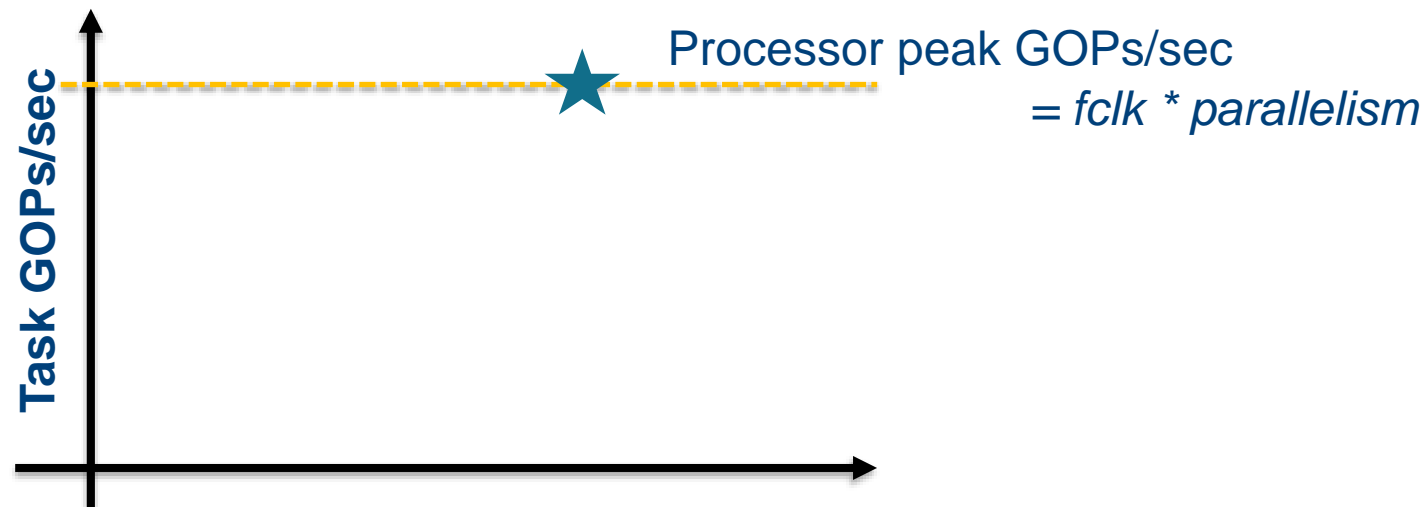
Strong vs Weak Scaling

- **Strong scaling:** problem size fixed
 - As in example

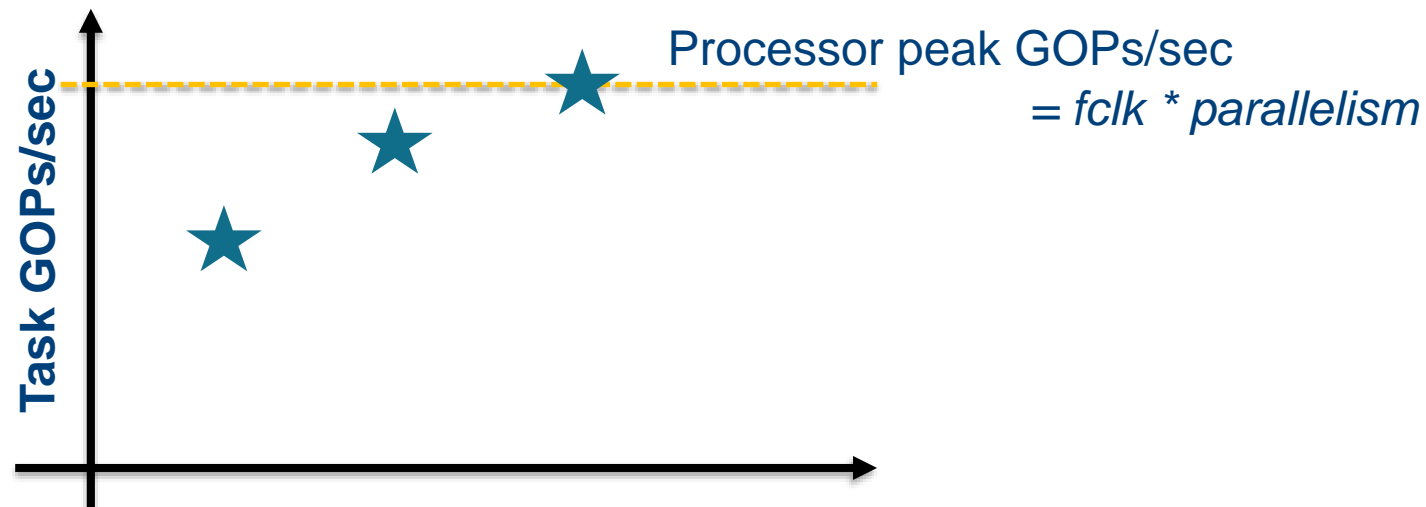
- **Weak scaling:** problem size proportional to number of processors
 - 10 processors, 10×10 matrix
 - Time = $20 \times t_{\text{add}}$
 - 100 processors, 32×32 matrix
 - Time = $10 \times t_{\text{add}} + 1000/100 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Constant performance in this example

- *Last week: Single issue functional parallelism: Pipelining*
- Instruction level parallelism
 - Static multiple-issue (VLIW)
 - Dynamic multiple issues (super-scalar)
- Thread level parallelism
- Core level parallelism
- **Challenges for parallelism?**
 - Amdahl's law
 - **Arithmetic intensity (memory bandwidth)**

- Performance metric of interest is actually executed useful operations/second (→ linearly related to seconds/task) = **GFLOPs/sec**, influenced by
 1. Compute capabilities



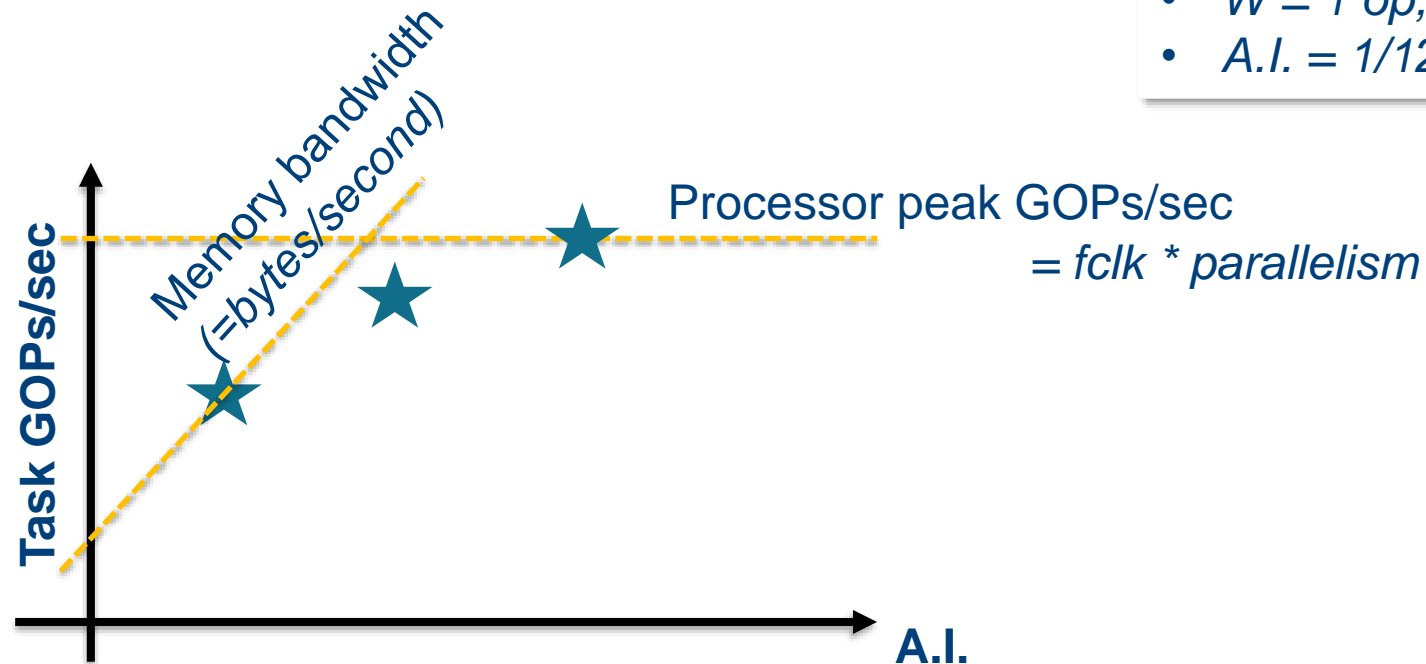
- Performance metric of interest is actually executed useful operations/second (→ linearly related to seconds/task) = **GFLOPs/sec**, influenced by
 1. Compute capabilities
 2. Memory bandwidth



- Arithmetic intensity of a kernel
 = *the ratio of the work W to the required memory traffic Q*
 - $A.I. = W/Q$
 - FLOPs / byte of memory traffic

Example: one 32-bit addition:

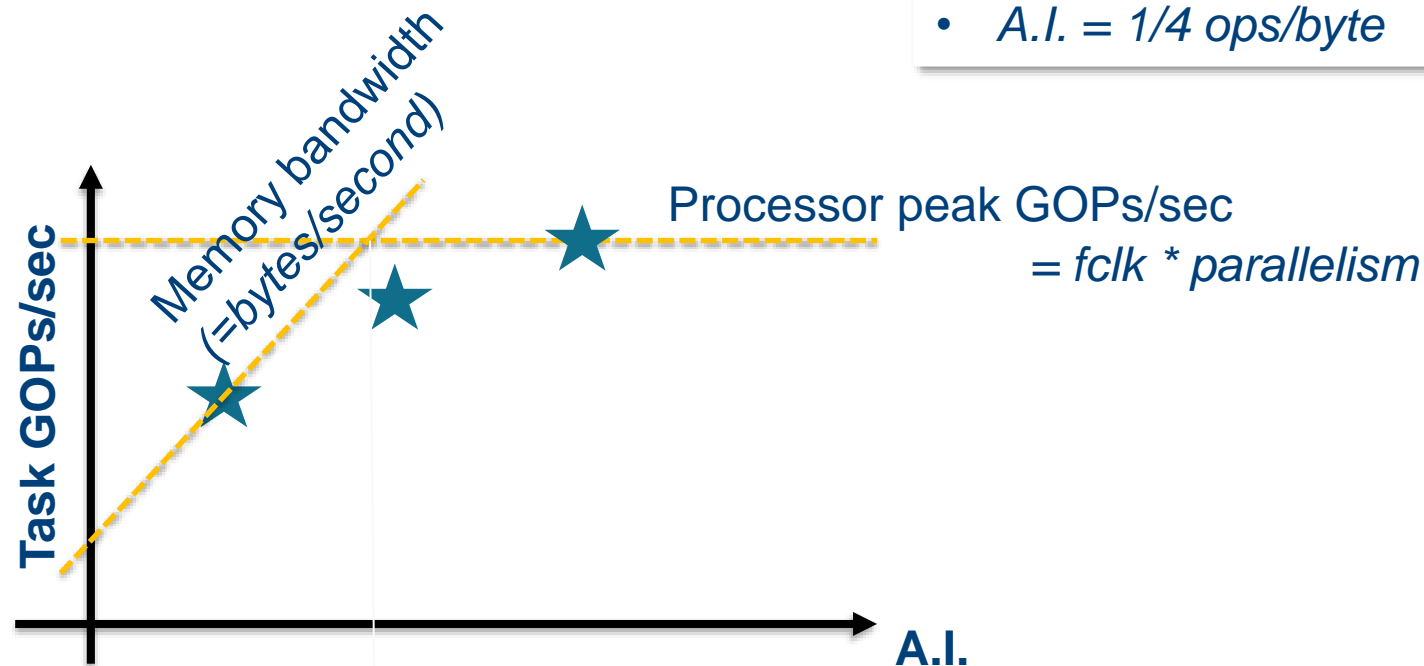
- 2 inputs, each 4 bytes
- 1 output, 4bytes
- $W = 1 \text{ op}$, $Q = 12 \text{ bytes}$
- $A.I. = 1/12$



- A.I. < cut point → communication constrained
- A.I. > cut point → computation constrained
- Where do you want to be in this graph?

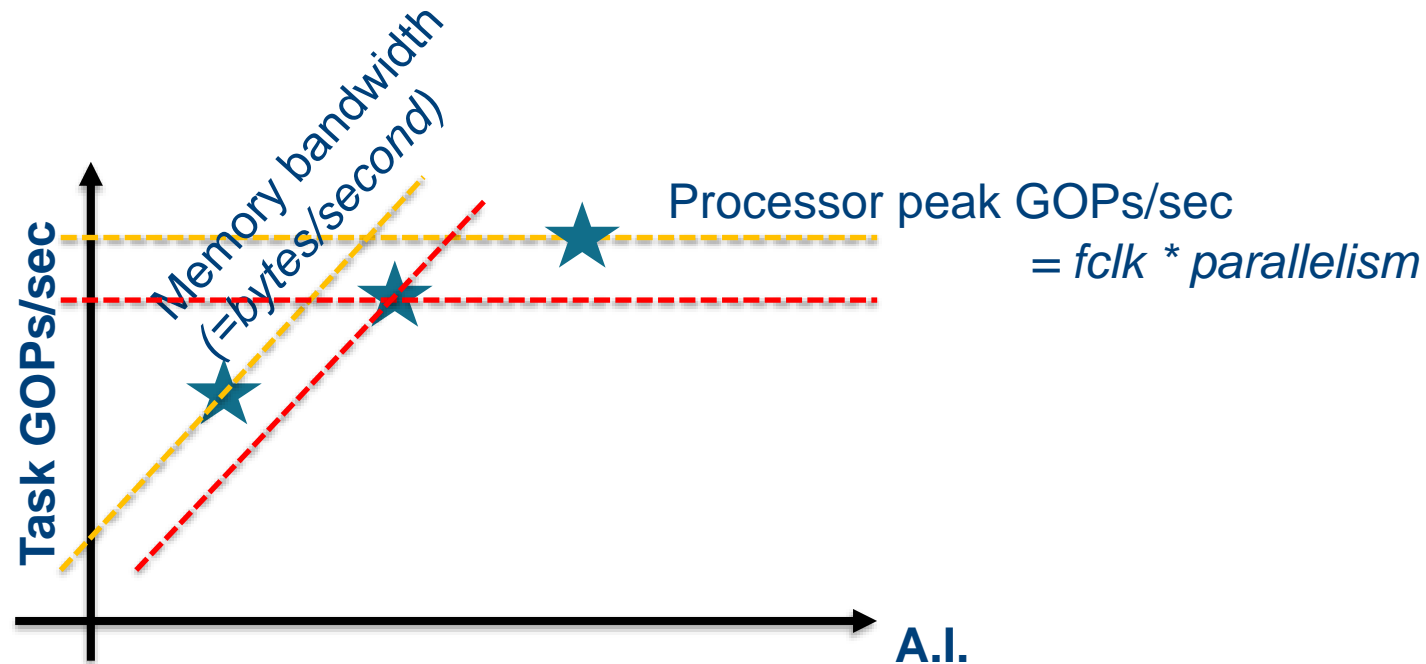
Example: image processing:

- Input image = 2MB, output image = 2MB
- $W = 1M$ op, $Q = 4MB$
- A.I. = 1/4 ops/byte



Sources of lost performance

1. Compute capabilities losses from **peak GFLOP/s**
 - NOPs due to hazards, insufficient code parallelism (e.g. SIMD), multi-core load imbalance,...
2. Memory capability losses from **peak bytes/sec**
 - Inefficient data fetching (cache miss i.s.o. prefetching)

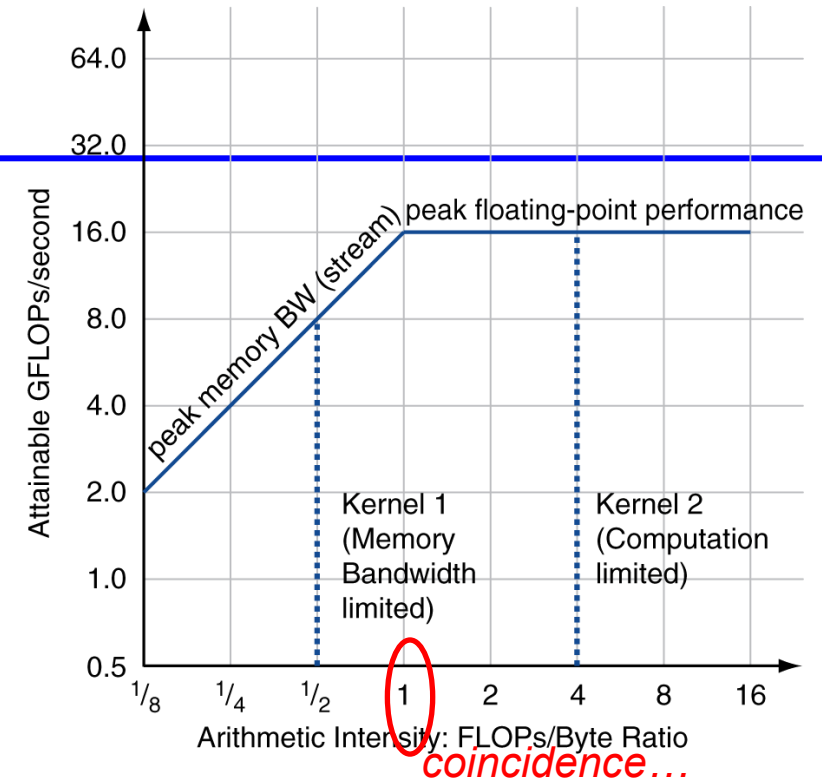


Creating a Roofline Diagram

Exercise: dual core processor,
with 2 issue slots, @4GHz clock,
with 16Gbyte/sec IO BW

For a given computer, determine
Peak GFLOPS (from data sheet)
Peak external memory bytes/sec

Attainable GFLOPs/sec
= Min (Peak Memory BW \times Arithmetic Intensity, Peak FP Performance)



- Processors combine a rich blend of parallelization strategies
- At different levels of abstraction:
 - Instruction level (VLIW, super-scalar)
 - Thread level
 - Core level
- Yet, performance does not scale up linearly
 - Dependencies limit achievable parallelism
 - Amdahl's law
 - Complexity leads to the power wall
 - Memory bandwidth
 - Arithmetic intensity; Roofline model

Representative exercises

Exercise1

A, B = vector [1, 1000]

Y1=Y2=Y3=0

For i=1..1000

Y1[i] = A[i] + B[i]

Y2 = Y2 + A[i]

Y3 = Y3 + A[i]*(Y3>0)

End

How much faster can Y1, resp. Y2, resp Y3 be computed on 10-way parallel processor?

Exercise 1: solution

- Y1 can be computed 10x faster ($f \sim 0$). $\rightarrow S=10$
- Y2 can be computed faster when rewriting sequential add to a parallel adder tree. f will not be 0, as at the end all the data has to be combined back together. But f can be small (e.g. 0.01)
 - E.g. first do 500 parallel adds, then 250 adds of these results, then 125 adds of the results, then 63, then 32, then 16, 8, 4, 2 and 1. Results ~ 10 last adds can not be fully parallelized, other 990 can.
 - So speedup $S = 10/(1+9*0.01) = 9.17$
- Y3 can not be computed faster as it really needs to be executed sequentially ($f \sim 1$). $\rightarrow S=1$

Exercise 2 (discuss solutions in online forum)

- Draw the hardware architecture of a VLIW processor with 3 issue slots, with following issue packet:
 - 1st issue slot: add / sub / addi / subi instructions
 - 2nd issue slot: beq / bneq instructions
 - 3rd issue slot: ld / st instructions

Exercise 3 (discuss solutions in online forum)

- What hazards do you find in the code underneath (control and data hazards & type of data hazards)? How many cycles does the code take on a 5-stage pipelined processor?
- Which hazards would be overcome by following techniques we saw. How much would it help (how many clock cycles would be saved)?
 - VLIW
 - Dynamic issuing with out-of-order execution without speculation
 - Dynamic issuing with out-of-order execution with speculation

Conventional MIPS code

	<code>addi</code>	<code>x4, x0, 128</code>	<code>;upper bound of what to load</code>
<code>loop:</code>	<code>ld</code>	<code>f2, 0 (x4)</code>	<code>;load x(i)</code>
	<code>ld</code>	<code>f4, 256 (x4)</code>	<code>;load y(i)</code>
	<code>mul</code>	<code>f4, f4, f2</code>	<code>;x(i) x y(i)</code>
	<code>sd</code>	<code>f4, 0 (x4)</code>	<code>;store into x(i)</code>
	<code>subi</code>	<code>x4, x4, 1</code>	<code>;decrement index</code>
	<code>bne</code>	<code>x4, x0, loop</code>	<code>;check if done</code>

Exercise 4 (discuss solutions in online forum)

- Construct the roofline plot of the following processor
 - Quad core processor, running at 2GHz, each enabling 4 instructions per cycle, and a 6 stage pipeline.
 - External memory bandwidth of 2GB/sec
- Are following operations memory or compute bounded on this processor? What is the maximum throughput they can achieve?
 1. Matrix-matrix multiplication of two 16x16 matrices, resulting in a 16x16 matrix (every data word is 64-bit)
 2. Denoising a camera image of 1M RGB pixels large, into a clean 1Mpixel RGB image, requiring 15,000,000 operations per image.