CIS 431/531
Intro to Parallel Computing

Lecture 3

Parallel Computer Architectures I

Logistics

Individual assignment repo - please create & share

Group project - please share the following info:

- Bitbucket repo
- Group name & members
- *Email me if you haven't found a group by Friday*

First assignment will be Oct 11 (Wednesday, week 3)

- Please work on your project & survey while the load is light

Previous Lecture

What is performance & how do you measure it?

Questions?

Quiz

Question 1 (9 minutes) -

- You have an application that takes **60 hours** to execute on a serial computer.
- Using **six processors**, you were able to achieve a speedup of **4x**.
- What portion (i.e., how many hours) of the application was **serial**, assuming the parallel portion was embarrassingly parallel?

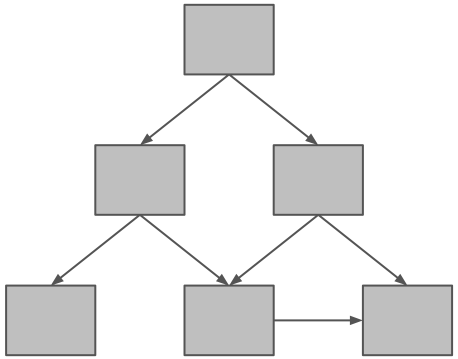
Hint: Amdahl's Law -> $S_p = 1 / (f + (1-f) / p)$

Quiz

Question 2 (6 minutes) -

Given the following DAG, what is

1. T_1 (i.e., how many units of time to complete the application using **one processor**)
2. T3 (i.e., how many units of time to complete the application using **three processors**)
3. T_∞(Span)



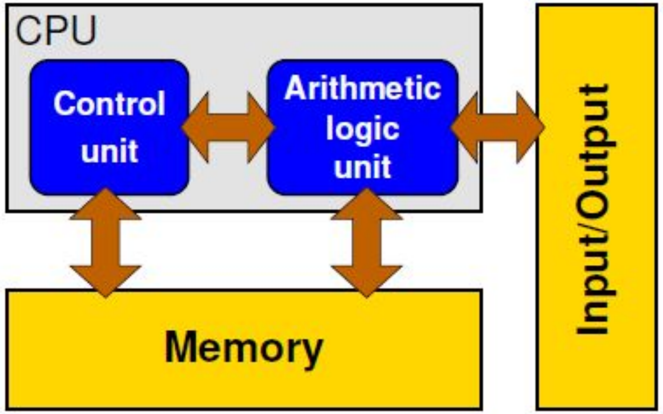
Computer 101

What is a computer?

Computer 101

Stored-program computer architecture (modern architecture)

- Conceived by Turing in 1936, built by Eckert and Mauchly in 1949 (Electronic Discrete Variable Automatic Computer, or **EDVAC**)
- von Neumann architecture - any stored-program architecture that instruction fetch and data operation **cannot occur simultaneously**



Architecture

Computer Architecture

- Microarchitecture - implementation of the processor
  - Pipelining, OOE, branch prediction, etc.
- Instruction Set Architecture (ISA) - syntax of the interface to the microarchitecture
  - Assembly instructions, registers, etc.

(Parallel) Architecture Types

Flynn’s Taxonomy (1966)

		Instruction stream	
		One	Many
Data stream	One	SISD traditional von Neumann single CPU computer	MISD ?
	Many	SIMD fine-grained data parallel computer (vector processor)	MIMD common multi-processor computer

(Parallel) Architecture Types

Flynn's Taxonomy (1966)

		Instruction stream	
		One	Many
Data stream	One	SISD traditional von Neumann single CPU computer	MISD fault-tolerant computer
	Many	SIMD fine-grained data parallel computer (vector processor)	MIMD common multi-processor computer

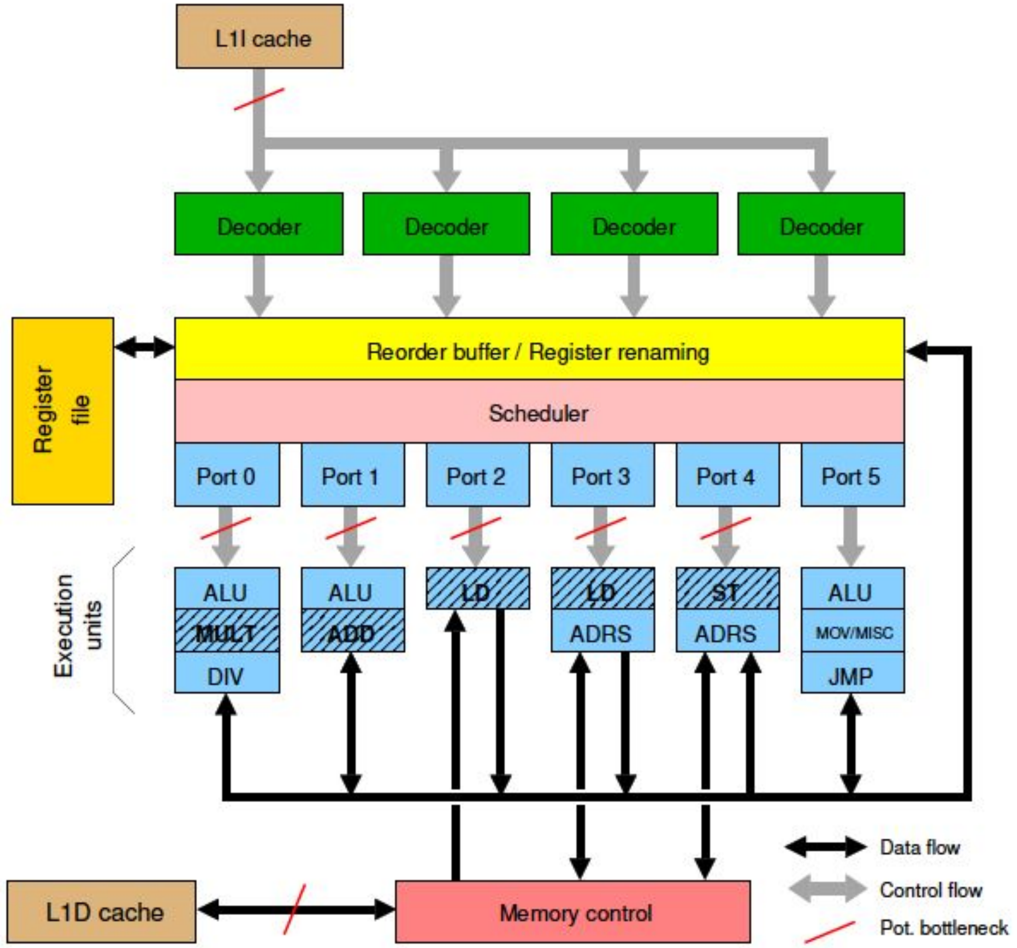
(Parallel) Architecture Types

Generally, most modern architectures do not fit perfectly into a single category.

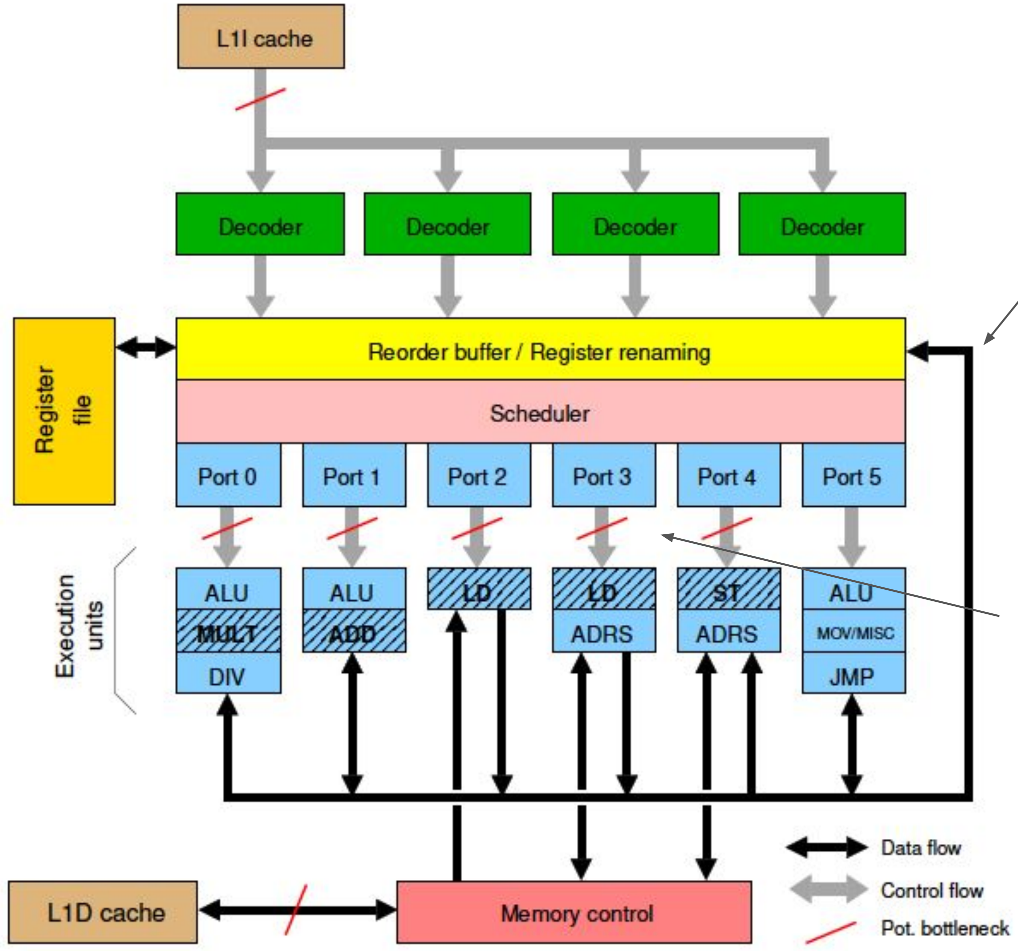
Most architectures fall under multiple categories, or use a combination of multiple paradigms.

- e.g., modern multi-core CPUs are MIMD, but also provide SIMD in their µarchitecture and ISA

Modern CPU Architecture



Modern CPU Architecture



Why

Why 2 LD but 1 ST unit?

Modern CPU Features (Overview)

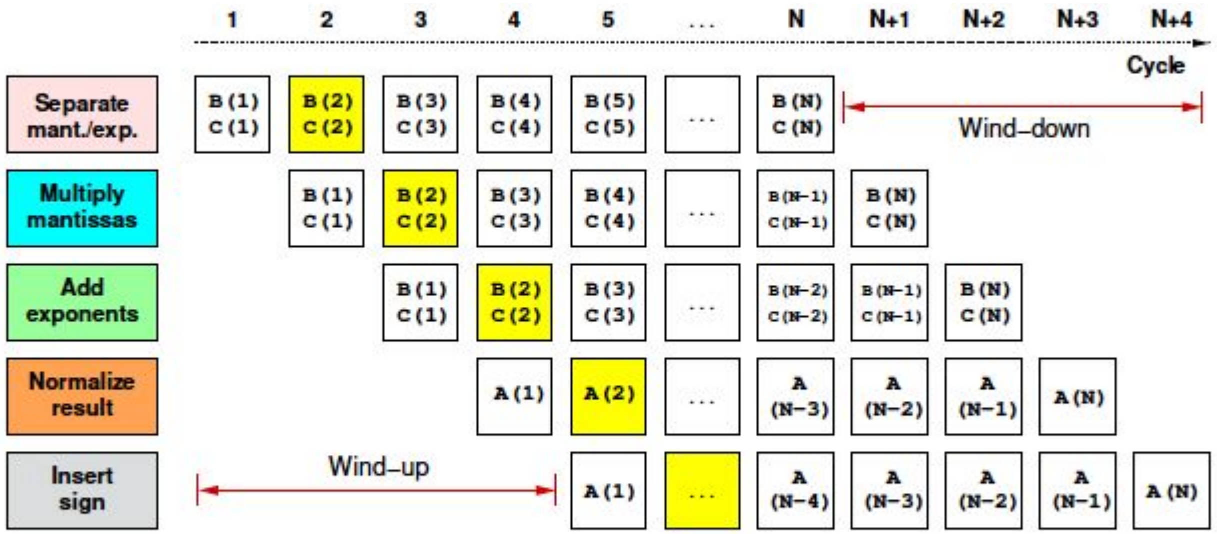
- Pipelining
- Superscalar & OOE
- Branch prediction
- Hardware (multi)threading
- SIMD
- Caches
- RISC vs. CISC

Pipelining

Dividing an instruction into smaller components executed by different functional units.

Increase throughput since the processor does not have to wait until an instruction has completed before starting execution of another instruction.

Pipelining (FP)



```
do i=1,N
      A(i) = B(i) * C(i)
enddo
```

Pipelining (FP)

If **no pipelining**, n instructions would take (**n x 5**) cycles (if each “step” takes 1 cycle).

If pipelined, n instructions would take (**n + 5 - 1**) cycles.

Assuming pipeline depth is m,

Speedup = ?

Pipelining (FP)

If **no pipelining**, n instructions would take (**n x 5**) cycles (if each "step" takes 1 cycle).

If pipelined, n instructions would take (**n + 5 - 1**) cycles.

Assuming pipeline depth is m,

Speedup = nm / (n + m - 1) = ~m (for n >> m)

What about **throughput (Instructions per cycle (IPC)**)?
No pipelining:
n / (nm) = 1/m
With pipelining:
n / (n + m - 1) = 1 / (1 + (m-1) / n) = ~1

Pipelining

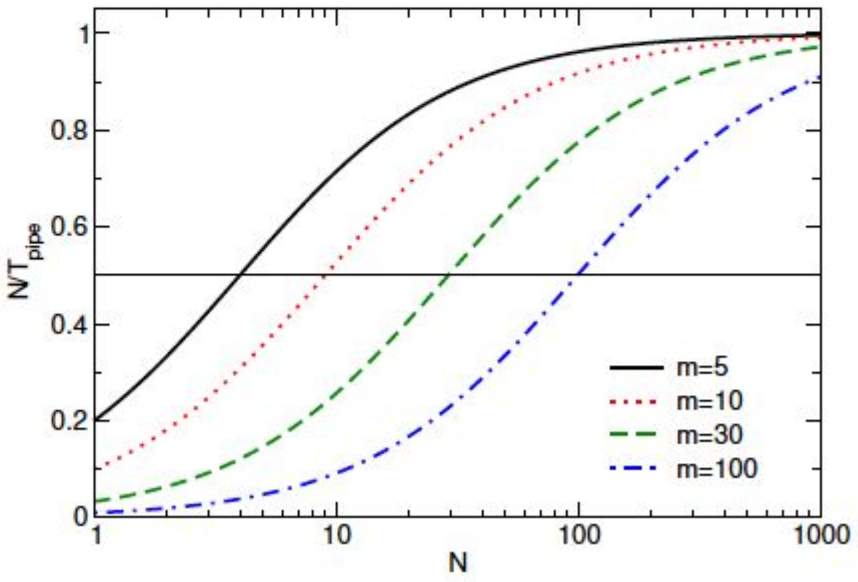
Speedup = nm / (n + m - 1) = ~m (for n >> m)

Does this mean that longer the pipeline, the better?

Hint: what is necessary for instructions to be issued every cycle?

Pipelining

Does this mean that longer the pipeline, the better?



Not necessarily - longer the pipeline, more **independent instructions** are required to amortize the "wind-up/down" phase.

Interleaving

Remember, instructions can only execute if the data to operate on is available in the register - data has to travel from the DRAM, through the various caches, and then to the register.

If the operands are not available, pipeline cannot function as intended.

Interleaving

```
1 do i=1,N
2       A(i) = s * A(i)
3 enddo
```

```
1 i = 1
2 LOAD R9 = s
3 loop: LOAD R1 = A(i)
4          R1 = MULT R1,R9 # "useful" work
5          STORE A(i) = R1
6          INC i # i=i+1
7          CMP i,N+1 # i.eq.N+1 ?
8          BNE loop # branch if not equal
```

The next instruction cannot be issued/scheduled until everything in the loop has completed

If LOAD takes 4 cycles, and MULT takes 2 cycles, it will take 6 cycles to reach instruction 5 (STORE)

Software Pipelining

```
i = 1
2 LOAD R9 = s
3 loop: LOAD R1 = A(i)
4       LOAD R2 = A(i+1)
5       LOAD R3 = A(i+2)
6       LOAD R4 = A(i+3)
7       R1 = MULT R1,R9
8       R2 = MULT R2,R9
9       STORE A(i) = R1
10      R3 = MULT R3,R9
11      STORE A(i+1) = R2
12      R4 = MULT R4,R9
13      STORE A(i+2) = R3
14      STORE A(i+3) = R4
15      IADD i,4
16      CMP i,N+1
17      BNE loop
```

After 4 cycles, A(i) is loaded

After 2 cycles, A(i) is ready for ST

Multiple instructions are ready, and can be scheduled in any order

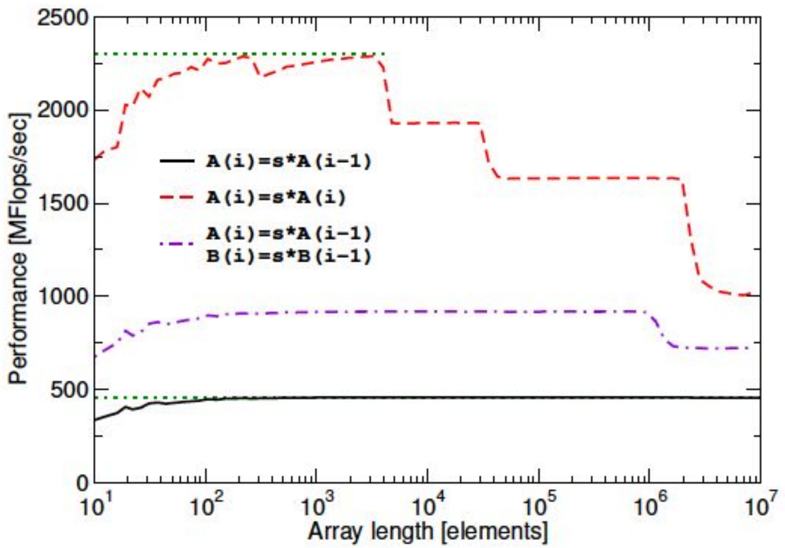
If LOAD takes 4 cycles, and MULT takes 2 cycles, it will take 6 cycles to reach instruction 5 (STORE)

Interleaving

What if there are loop-carried dependencies (i.e., iteration i depends on results from iteration j != i)?

```
Real dependency

1 do i=2,N
2     A(i)=s*A(i-1)
3 enddo
```



5 cycles per loop

Interleaving

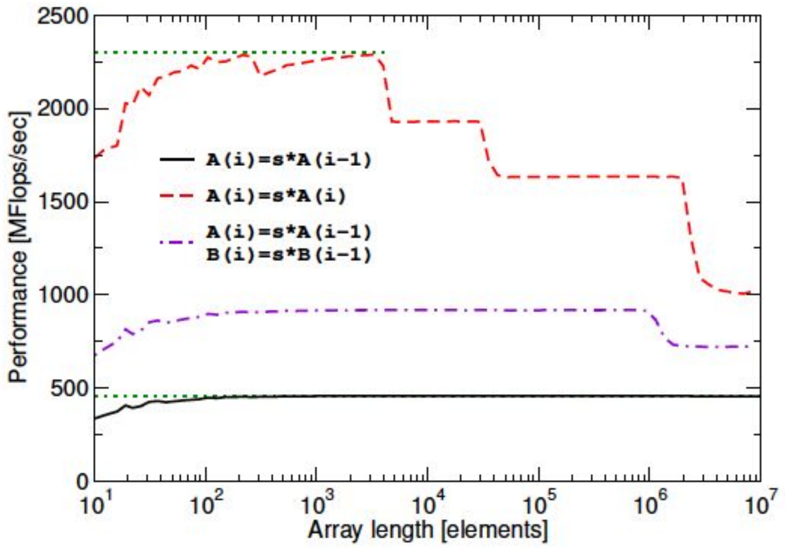
What if there are loop-carried dependencies (i.e., iteration i depends on results from iteration j != i)?

Real dependency

```
1 do i=2,N
2      A(i)=s*A(i-1)
3 enddo
```

Pseudo-dependency

```
1 do i=1,N-1
2      A(i)=s*A(i+1)
3 enddo
```

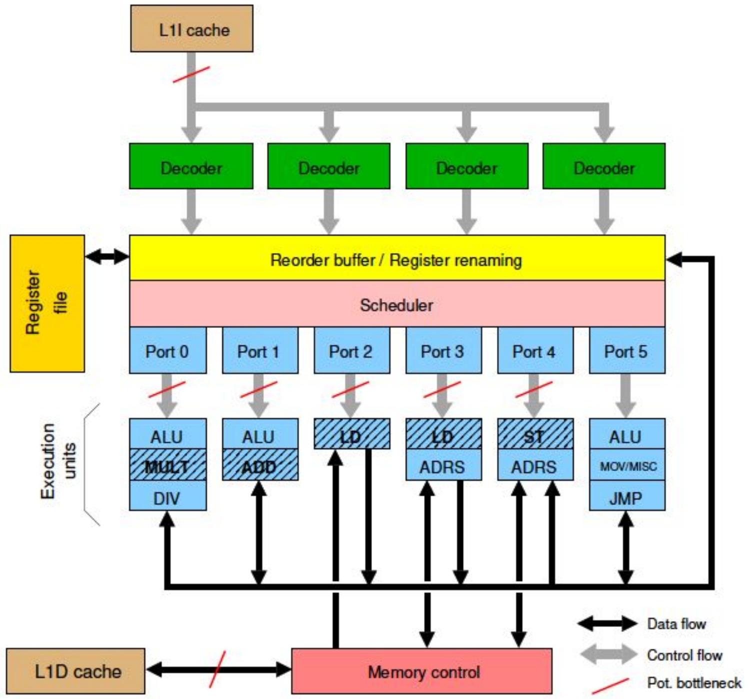


5 cycles per loop

Questions?

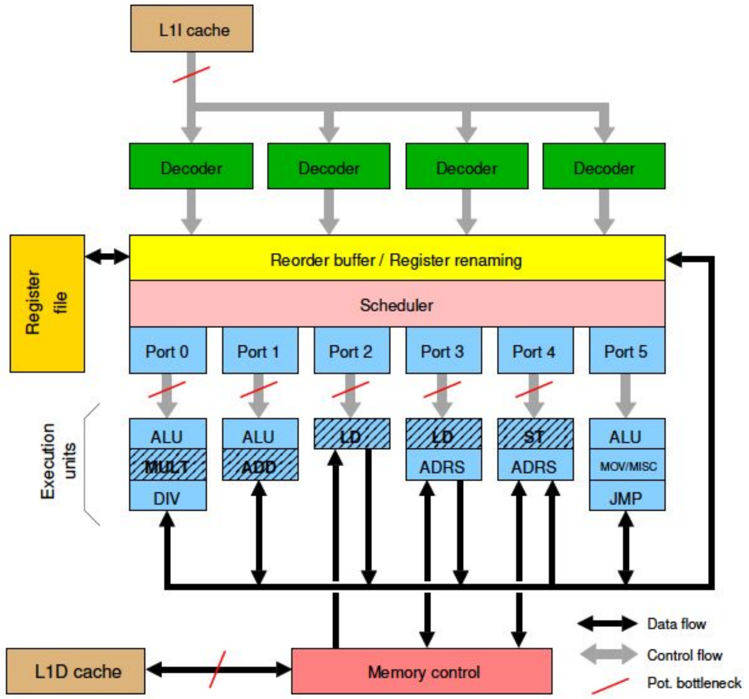
Superscalar and OOE

Superscalar execution - a processor is capable of executing more than one instruction at the same time (**ILP - instruction level parallelism**)



Superscalar and OOE

What is the maximum number of **sustained** IPC this system can deliver?



Superscalar and OOE

Why did I explain superscalar and OOE together?

- Well, to execute multiple instructions at the same time, simple in-order execution is typically incapable of providing enough independent instructions to fill the pipeline.
- Therefore, we have a reordering buffer that analyzes the instruction streams to find independent instructions and schedule ready-to-execute instructions even if it came later - i.e., out-of-order execution.

Superscalar and OOE

Reordering buffer

- Scoreboarding
- Tomasulo’s algorithm

Hazards

Hazards

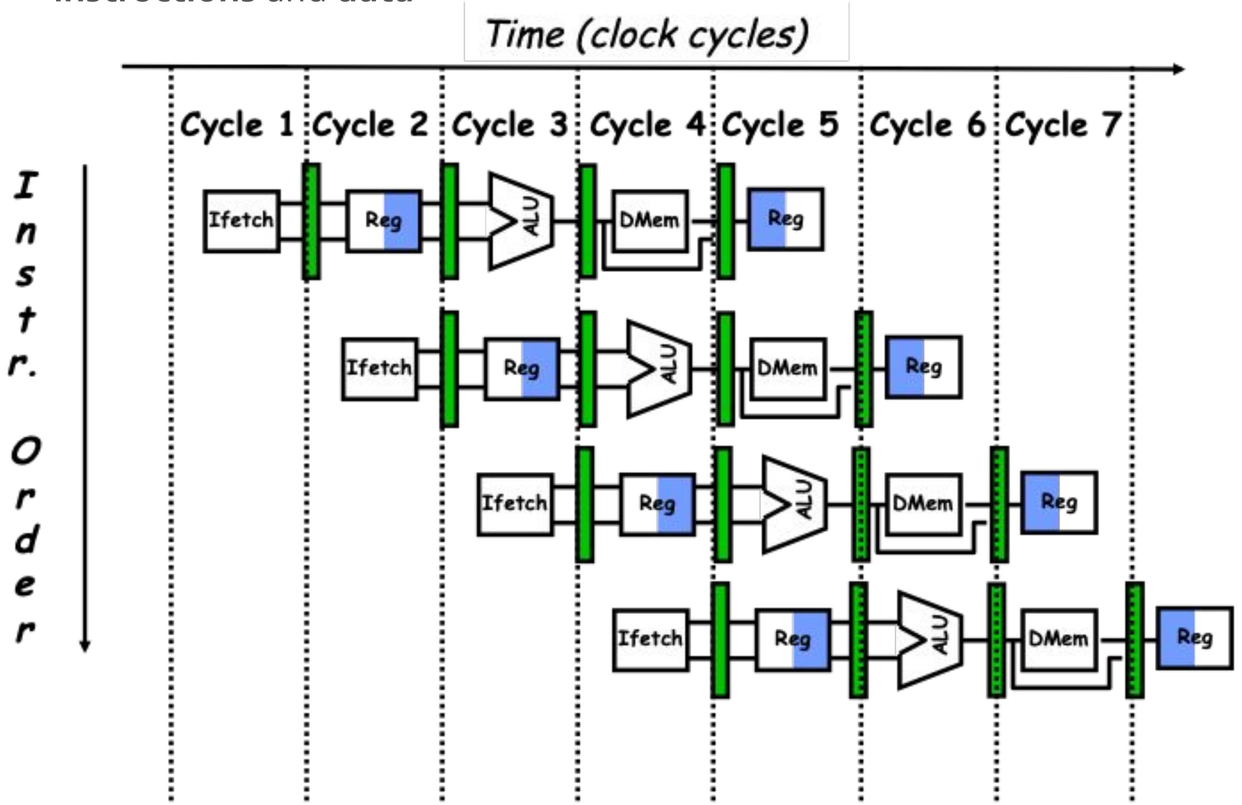
- Structural hazards
  - Occurs when a part of processor's hardware is needed by two or more instructions at the same time
- Control hazards
  - Branches - the architecture does not know which instruction to fetch next
- Data hazards
  - Read after write (RAW) - true dependency
    - i2 reads a source before i1 writes to it
    - `i1. R2 <- R1 + R3`
    - `i2. R4 <- R2 + R3`
  - Write after read (WAR) - anti-dependency
    - i2 writes to a destination before i1 reads it
    - `i1. R4 <- R1 + R5`
    - `i2. R5 <- R1 + R2`
  - Write after write (WAW) - output dependency
    - i2 writes to a destination before it is written by i1
    - `i1. R2 <- R4 + R7`

      …
    - `i2. R2 <- R1 + R3`

Hazards

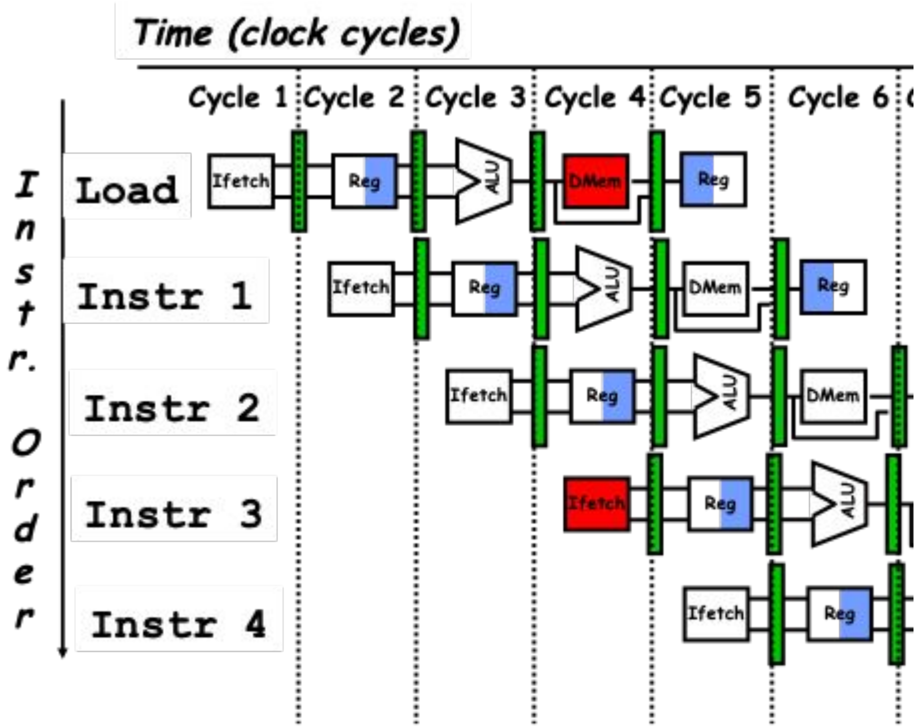
Assume single-issue, in-order, and 5 stage pipeline, 1 unit type each

Assume there is only 1 memory unit and it services both **read and write** of **instructions** and **data**



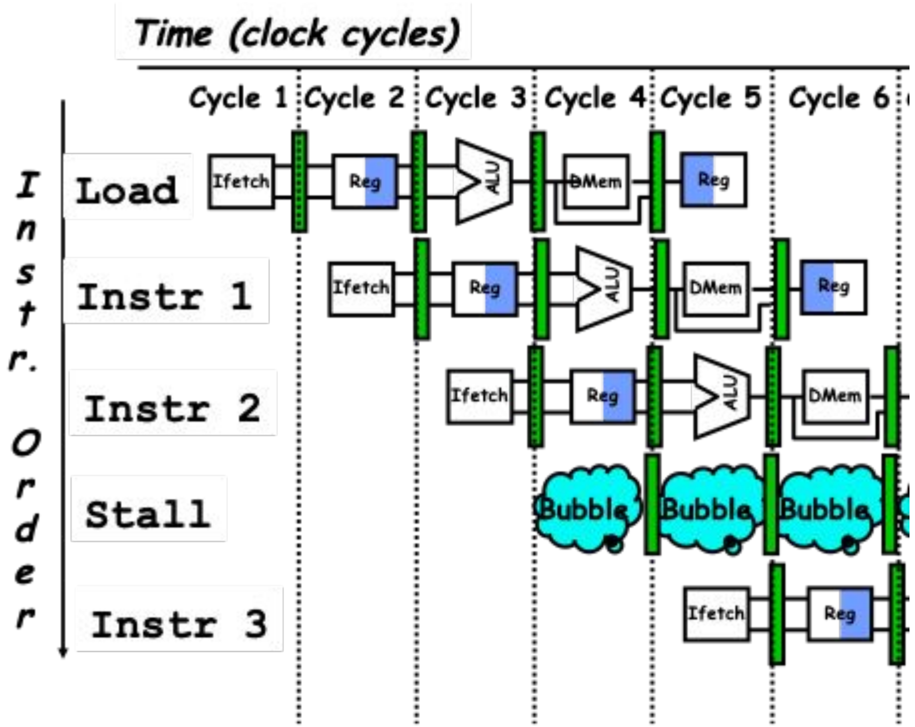
Structural Hazard

Only 1 memory unit, and both instructions need to use it to fetch data and instruction



Structural Hazard

Insert “bubbles” (i.e., stalls) in the pipeline.



Branch Hazard

Branch prediction

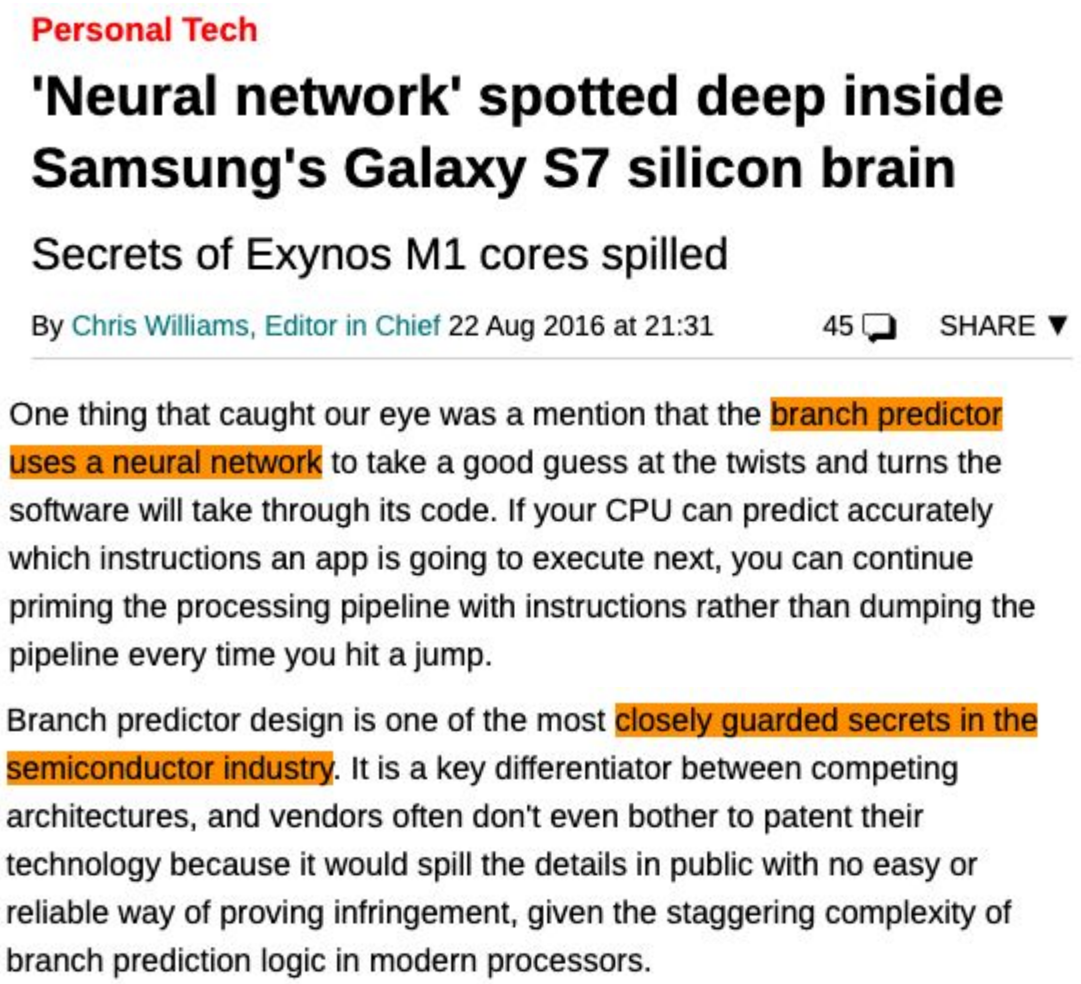
- The processor tries to “guess” whether the branch will be taken or not.
- If the prediction was incorrect (i.e., mispredicted), then the pipeline needs to be flushed - very expensive.
- Why does branch prediction work? Shouldn’t it be 50/50? (Hint: what is a big source of branches?)

Branch Hazard

Branch prediction

- The processor tries to “guess” whether the branch will be taken or not.
- If the prediction was incorrect (i.e., mispredicted), then the pipeline needs to be flushed - very expensive.
- Loops are a big source of branches - branches are **taken more often than not**.

Branch Prediction



Personal Tech

'Neural network' spotted deep inside Samsung's Galaxy S7 silicon brain

Secrets of Exynos M1 cores spilled

By Chris Williams, Editor in Chief 22 Aug 2016 at 21:31 45 SHARE ▼

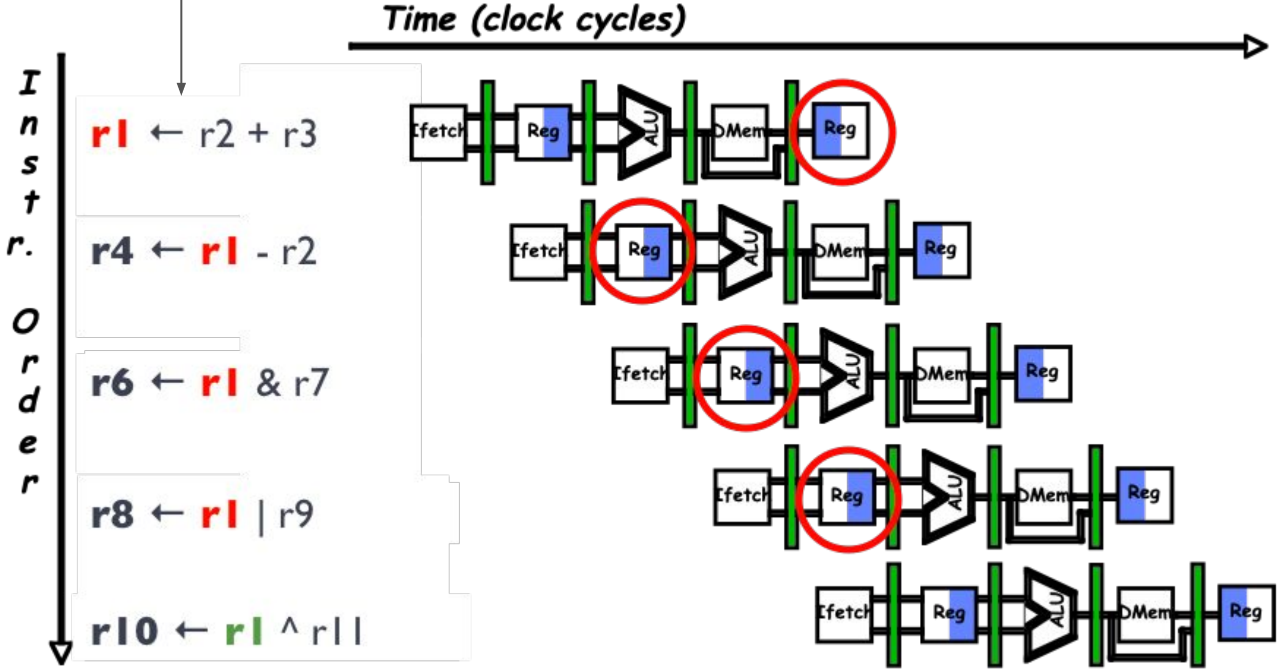
One thing that caught our eye was a mention that the branch predictor uses a neural network to take a good guess at the twists and turns the software will take through its code. If your CPU can predict accurately which instructions an app is going to execute next, you can continue priming the processing pipeline with instructions rather than dumping the pipeline every time you hit a jump.

Branch predictor design is one of the most closely guarded secrets in the semiconductor industry. It is a key differentiator between competing architectures, and vendors often don't even bother to patent their technology because it would spill the details in public with no easy or reliable way of proving infringement, given the staggering complexity of branch prediction logic in modern processors.

Questions?

Data Hazards (RAW)

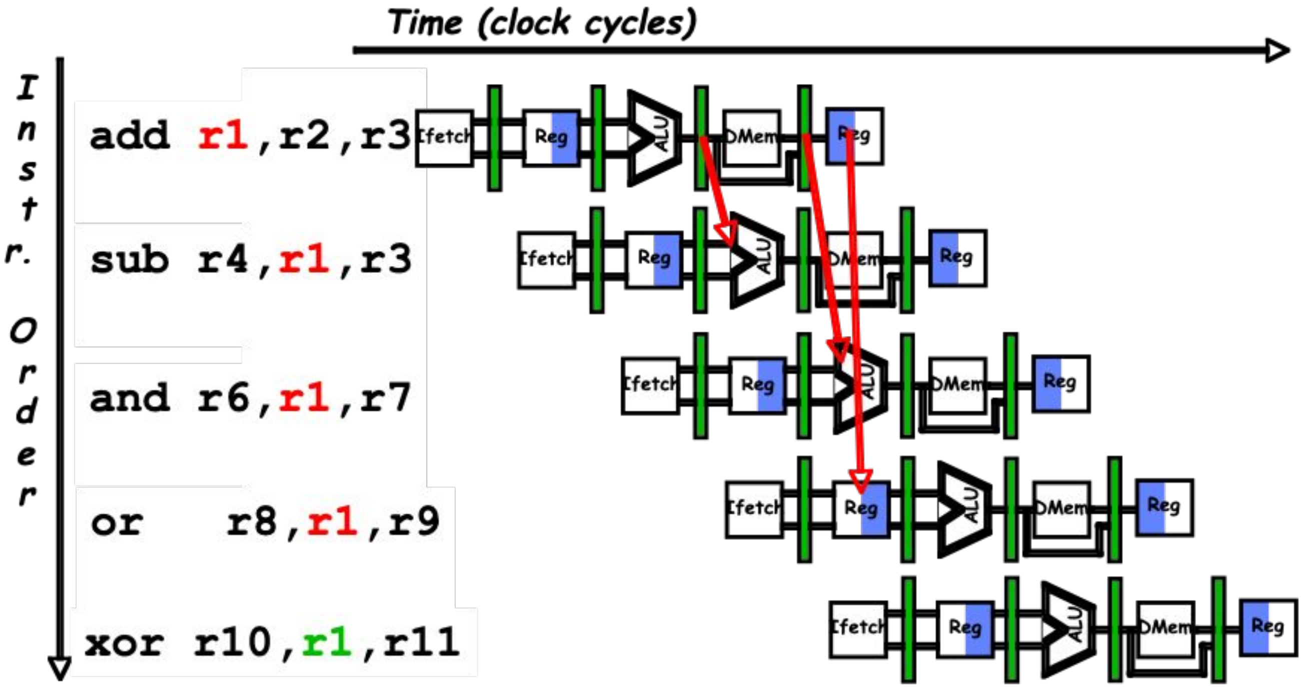
RAW



x

Data Hazards (RAW)

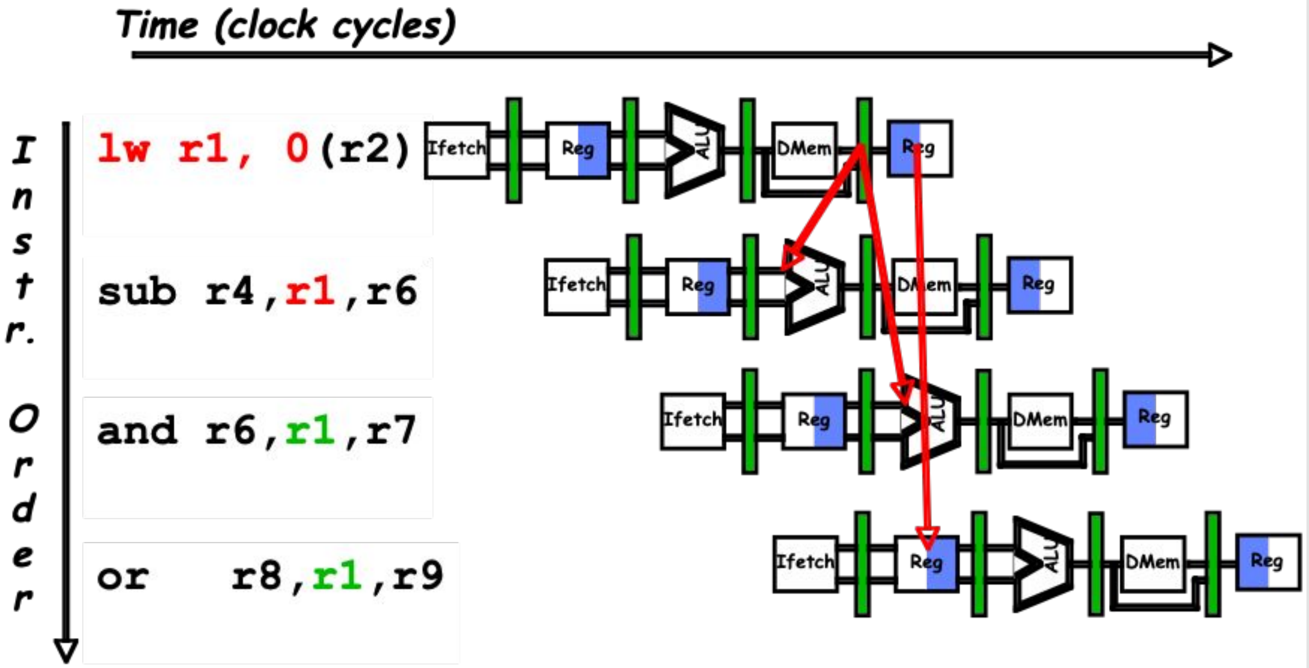
Operand Forwarding



```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```

Data Hazards

Does not always work…



Scoreboarding and Tomasulo’s Algorithm

OOE helps “alleviate” these data hazards, so that more independent instructions can be identified and issued.

Scoreboarding simply keeps track of instruction dependencies to determine whether an instruction can start executing

It does **not eliminate** dependencies

Tomasulo’s algorithm uses scoreboarding with register renaming/coloring to eliminate WAR and WAW dependencies.

WAR and WAW can be overcome if we change the source or destination registers for one of the instructions that have dependencies.

WAR

```
i1. R4 <- R1 + R5                ??
i2. R5 <- R1 + R2
```

WAW

```
i1. R2 <- R4 + R7                ??
i2. R2 <- R1 + R3
```

Scoreboarding and Tomasulo's Algorithm

OOE helps "alleviate" these data hazards, so that more independent instructions can be identified and issued.

Scoreboarding simply keeps track of instruction dependencies to determine whether an instruction can start executing

It does **not eliminate** dependencies

Tomasulo's algorithm uses scoreboarding with register renaming/coloring to eliminate WAR and WAW dependencies.

WAR and WAW can be overcome if we change the source or destination registers for one of the instructions that have dependencies.

WAR

```
i1. R4 <- R1 + R5
i2. R5 <- R1 + R2
```

```
i1. R4 <- R1 + R5
i2. R6 <- R1 + R2
```

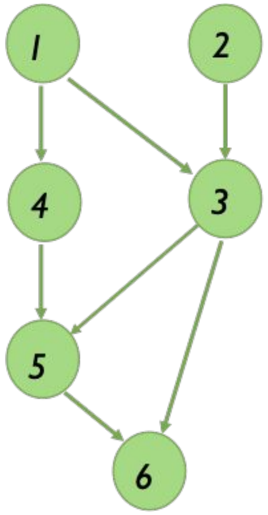
WAW

```
i1. R2 <- R4 + R7
i2. R2 <- R1 + R3
```

```
i1. R2 <- R4 + R7
i2. R6 <- R1 + R3
```

Scoreboarding and Tomasulo’s Algorithm

					latency
1	LD	F2 ←	34(R2)		1
2	LD	F4 ←	45(R3)		long
3	MUL	F6 ←	F4,	F2	3
4	SUB	F8 ←	F2,	F2	1
5	DIV	F4 ←	F2,	F8	4
6	ADD	F10 ←	F6,	F4	1



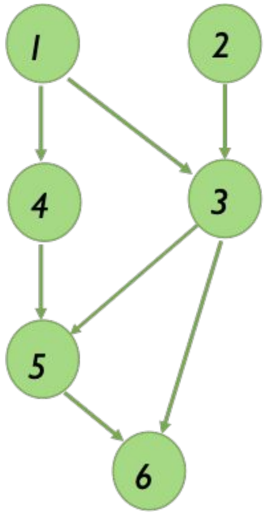
$$\begin{tikzcd}
1 \arrow[d] \arrow[dr] & 2 \arrow[d] \\
4 \arrow[d] & 3 \arrow[dl] \arrow[ddl] \\
5 \arrow[d] & \\
6 &
\end{tikzcd}$$

In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$
Out-of-order:

Instruction 1 has finished executing, and 2 begins execution

Scoreboarding and Tomasulo’s Algorithm

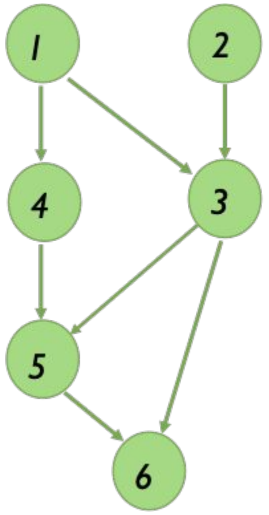
					latency
1	LD	F2 ←	34(R2)		*1*
2	LD	F4 ←	45(R3)		*long*
3	MUL	F6 ←	F4,	F2	*3*
4	SUB	F8 ←	F2,	F2	*1*
5	DIV	F4 ←	F2,	F8	*4*
6	ADD	F10 ←	F6,	F4	*1*



```
In-order:     1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . 3 5 . . . 5 6 6
```

Scoreboarding and Tomasulo’s Algorithm

					latency
1	LD	F2 ←	34(R2)		*1*
2	LD	F4 ←	45(R3)		*long*
3	MUL	F6 ←	F4,	F2	*3*
4	SUB	F8 ←	F2,	F2	*1*
5	DIV	F4 ←	F2,	F8	*4*
6	ADD	F10 ←	F6,	F4	*1*

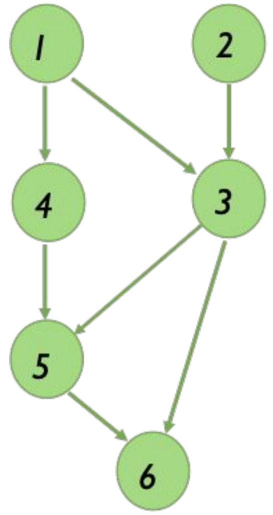


In-order: 1 (2,$\underline{1}$) . . . . . . $\underline{2}$ 3 4 $\underline{4}$ $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

Out-of-order: 1 (2,$\underline{1}$) 4 $\underline{4}$ . . . . $\underline{2}$ 3 . . $\underline{3}$ 5 . . . $\underline{5}$ 6 $\underline{6}$

Scoreboarding and Tomasulo’s Algorithm

					latency
1	LD	F2 ←	34(R2)		*1*
2	LD	F4 ←	45(R3)		*long*
3	MUL	F6 ←	F4,	F2	*3*
4	SUB	F8 ←	F2,	F2	*1*
5	DIV	F4 ←	F2,	F8	*4*
6	ADD	F10 ←	F6,	F4	*1*

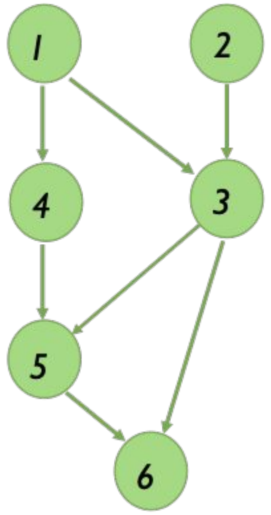


```
In-order:     1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . 3 5 . . . 5 6 6
```

In this case, OOE using scoreboarding does not help that much - both in-order and OO finish at the same time.

Scoreboarding and Tomasulo’s Algorithm

					latency
1	LD	F2 ←	34(R2)		1
2	LD	F4 ←	45(R3)		long
3	MUL	F6 ←	F4,	F2	3
4	SUB	F8 ←	F2,	F2	1
5	DIV	F4 ←	F2,	F8	4
6	ADD	F10 ←	F6,	F4	1

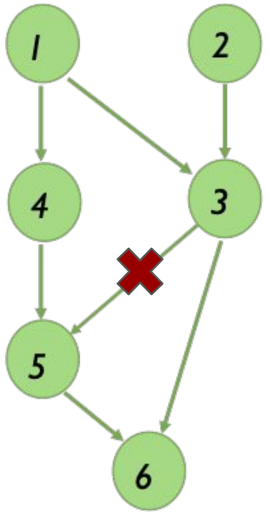


```
In-order:     1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . 3 5 . . . 5 6 6
```

In this case, OOE using scoreboarding does not help that much - both in-order and OO finish at the same time.

Scoreboarding and Tomasulo's Algorithm

					latency
1	LD	F2 ←	34(R2)		*1*
2	LD	F4 ←	45(R3)		*long*
3	MUL	F6 ←	F4,	F2	3
4	SUB	F8 ←	F2,	F2	*1*
5	DIV	F4' ←	F2,	F8	4
6	ADD	F10 ←	F6,	F4'	*1*



```
In-order:     1 (2,1) . . . . . . 2 3 4 4 3 5 . . . 5 6 6
Out-of-order: 1 (2,1) 4 4 . . . . 2 3 . . 3 5 . . . 5 6 6
```

Tomosulo's algorithm can "rename" the registers in the hardware so that instruction 5 no longer depends on instruction 3.

Questions?

Modern CPU Features (Overview)

Pipelining

Superscalar & OOE

Branch prediction

Hardware (multi)threading

SIMD

Caches

RISC vs. CISC

Threads and Processes

Process - collection of resources required to execute a program

- Virtual address space, executable code, etc.

Thread - unit of execution/resource utilization

- Program counter, stack, a set of registers, etc.
- Typically, multiple threads are executing within a process

Context switching - when you want to switch from executing one thread to another

- Expensive for threads
- More expensive for processes

Hardware Threading

Hardware (multi)threading.

- **Simultaneous multithreading (SMT).**
- Intel - hyper threading (HT).
- AMD - clustered multithreading (CMT).

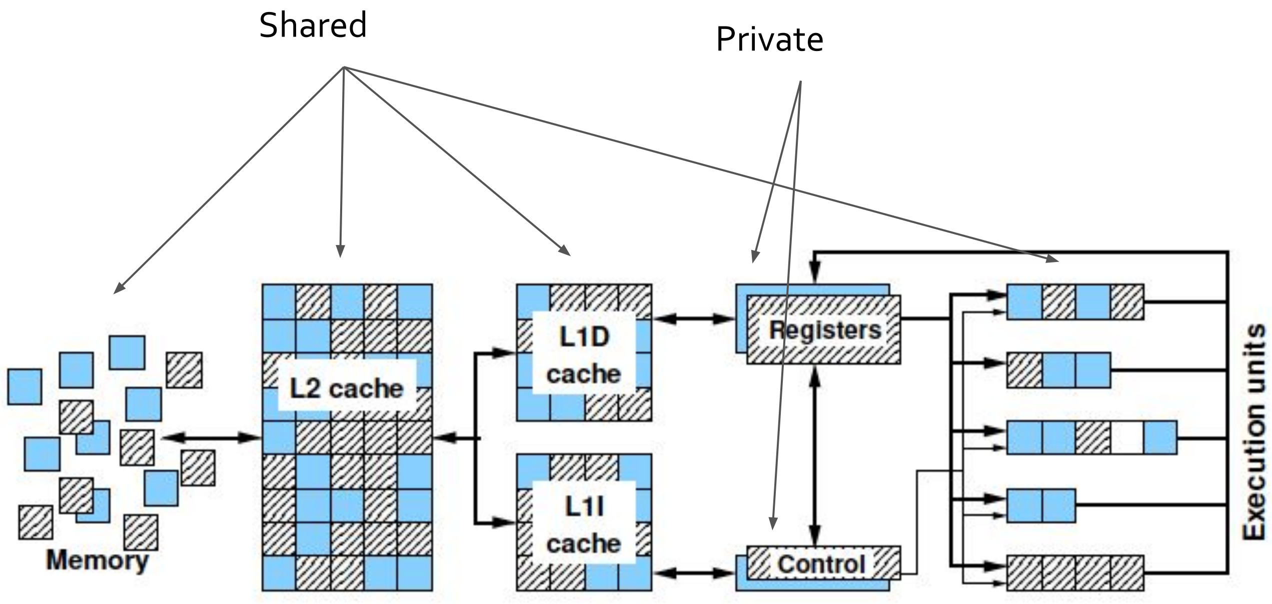
Each SMT thread maintain its own architectural “state.”

- Data registers.
- Control registers (e.g., stack pointer, instruction pointer, etc.).
- Switching from one thread to another no long requires expensive context switching

They still **share** execution resources

- Execution pipeline
- Cache

Hardware Threading



Hardware Threading

SMT **increases ILP** by adding another thread of execution - additional thread may come from a different application.

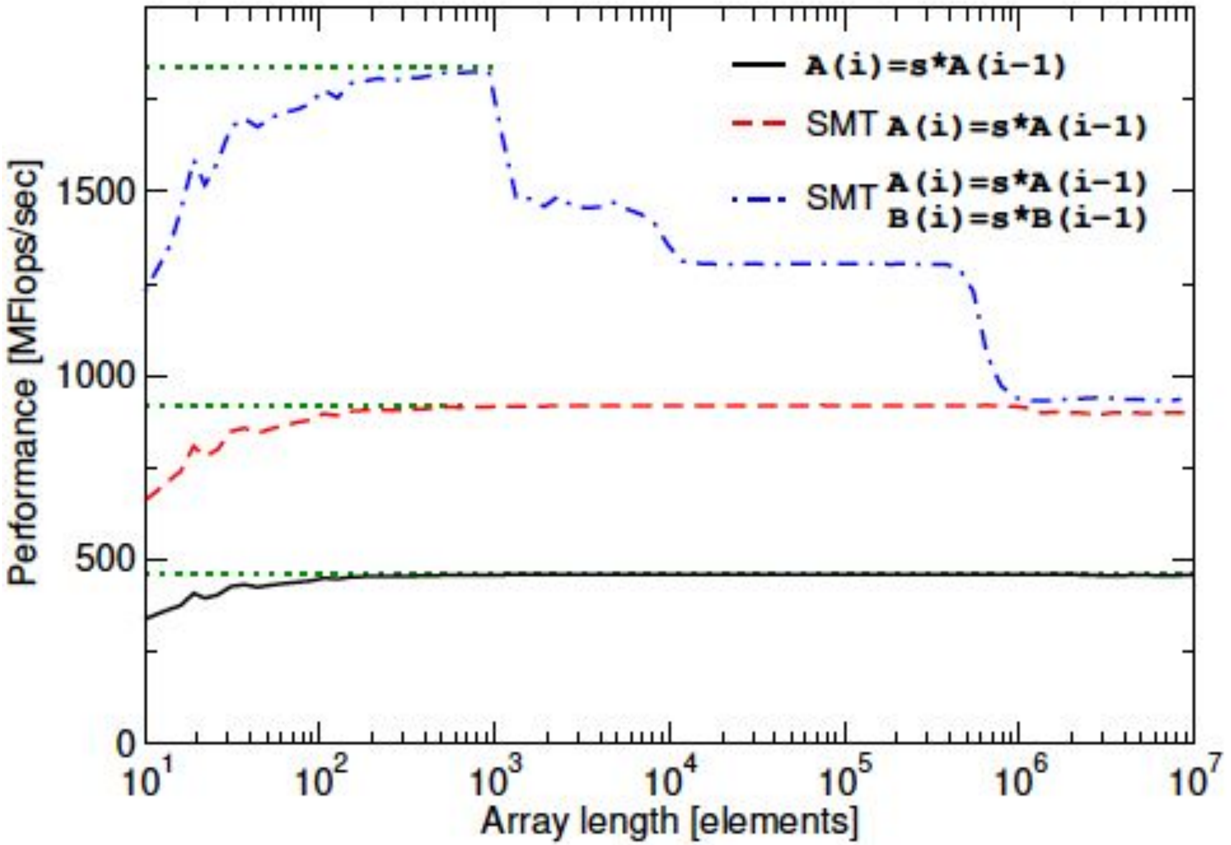
However, it **may** decrease performance as well

- One thread has enough ILP to fill the pipeline - now, fewer resources (e.g., registers, cache) are available for this tread if SMT is used.

Some modern architectures **require** SMT to fully utilize the pipeline

- Intel Xeon Phi (*RIP*)
- IBM Power8/9
    - One thread gets 64 entries in the fetch buffer
    - Two thread gets 128 entries
    - 4-8 threads gets 128 entries

Hardware Threading

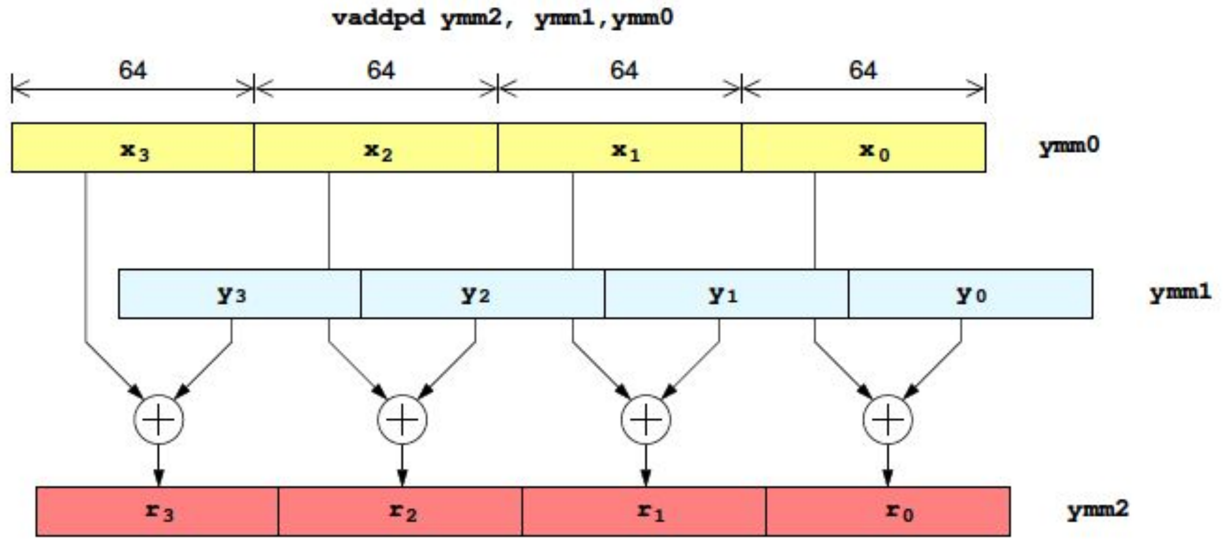


SIMD

Single Instruction Multiple Data

- First implemented in **vector** processors
- Also referred to as SIMD **vectorization**
- Present in most recent processors
    - Intel - SSE/AVX
    - AMD - 3DNow! (then SSE)
    - Arm - NEON
- Compiler can sometimes figure this out automatically (they are pretty good at it now)
- You can use intrinsics (special C/C++ instructions that mimic SIMD assembly instructions) - they are just “hints” so the compiler does not guarantee SIMD vectorization.

SIMD


vaddpd ymm2, ymm1,ymm0
64
64
64
64
x3
x2
x1
x0
ymm0
y3
y2
y1
y0
ymm1
r3
r2
r1
r0
ymm2

(Intel) CPU Programming

Intel provides a DETAILED guide on how to program their CPUs using assembly/intrinsic instructions

Do not read it (it's ~5000 pages). However, it could be a good place lookup how certain instructions work.

Intel Software Developer's Manual

Questions?