

Topic 4

Advanced Processors I

Xinfei Guo
xinfei.guo@sjtu.edu.cn

May 25th, 2022



T4 learning goals

- Advanced Processors
 - Section I: Superpipelined & Superscalar Pipelines (in-order)
 - Section II & III: Out-of-order (OoO) Pipelines

Recall the 1st lecture: Parallelism

- **So far: pipeline-level parallelism**
 - Work on execute of one instruction in parallel with decode of next
- **Next: instruction-level parallelism (ILP)**
 - Execute multiple independent instructions fully in parallel
- **Can do more:**
 - Static & dynamic scheduling
 - Extract much more ILP
 - Data-level parallelism (DLP)
 - Single-instruction, multiple data (one insn., four 64-bit adds)
 - Thread-level parallelism (TLP)
 - • Multiple software threads running on multiple cores

Recap: Non-pipelined and Pipelined Processors

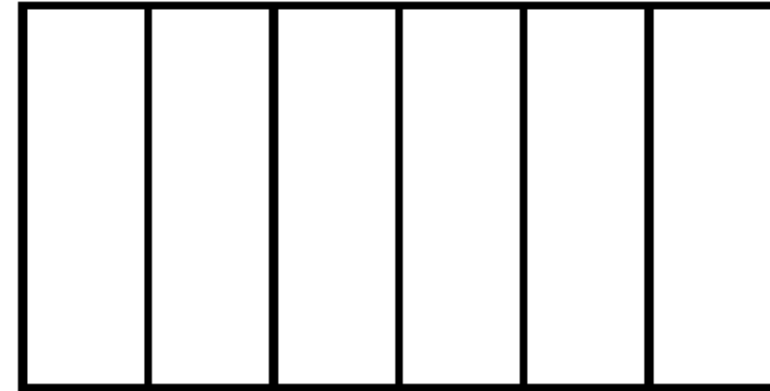
Non-pipelined processor



$IPC = 1$, Clock Period = T

(Note: IPC may be less than one if we assume we sometimes need to access a slow main memory.)

Pipelined processor



S – number of pipeline stages

$IPC \leq 1$ (aka “Flynn Bottleneck”)

Clock Period = $T/S + C$

(C = pipelining overhead)

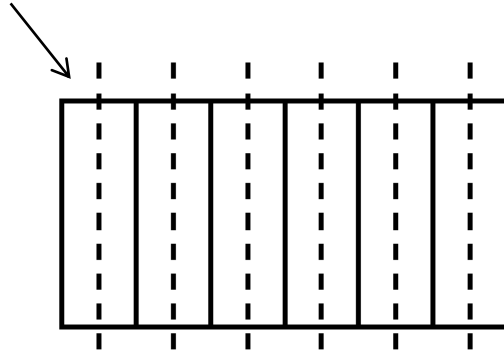
This is a **scalar pipeline**.

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - Multiple **issue**
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz **4-way** multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - But dependencies reduce this in practice

Superpipelined and Superscalar Processors

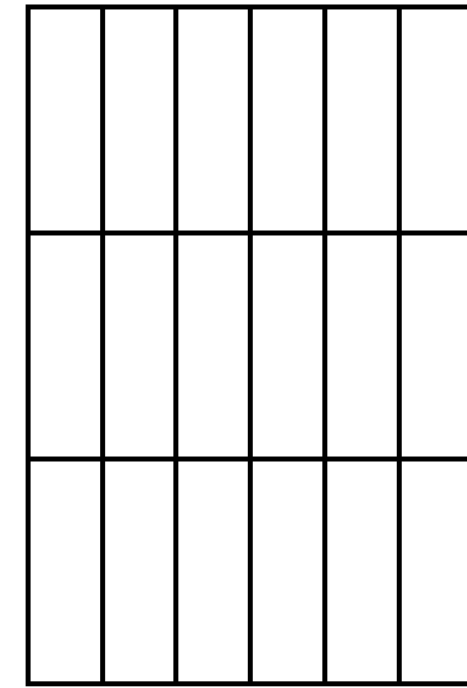
M sub-stages per stage



A Superpipelined Processor

The S pipeline stages (here $S = 6$) are further divided into M sub-stages (here $M=2$).

This processor executes M instructions during each of the original pipelined processor's clock periods. Its clock is M times faster.



Superscalar
Degree = P

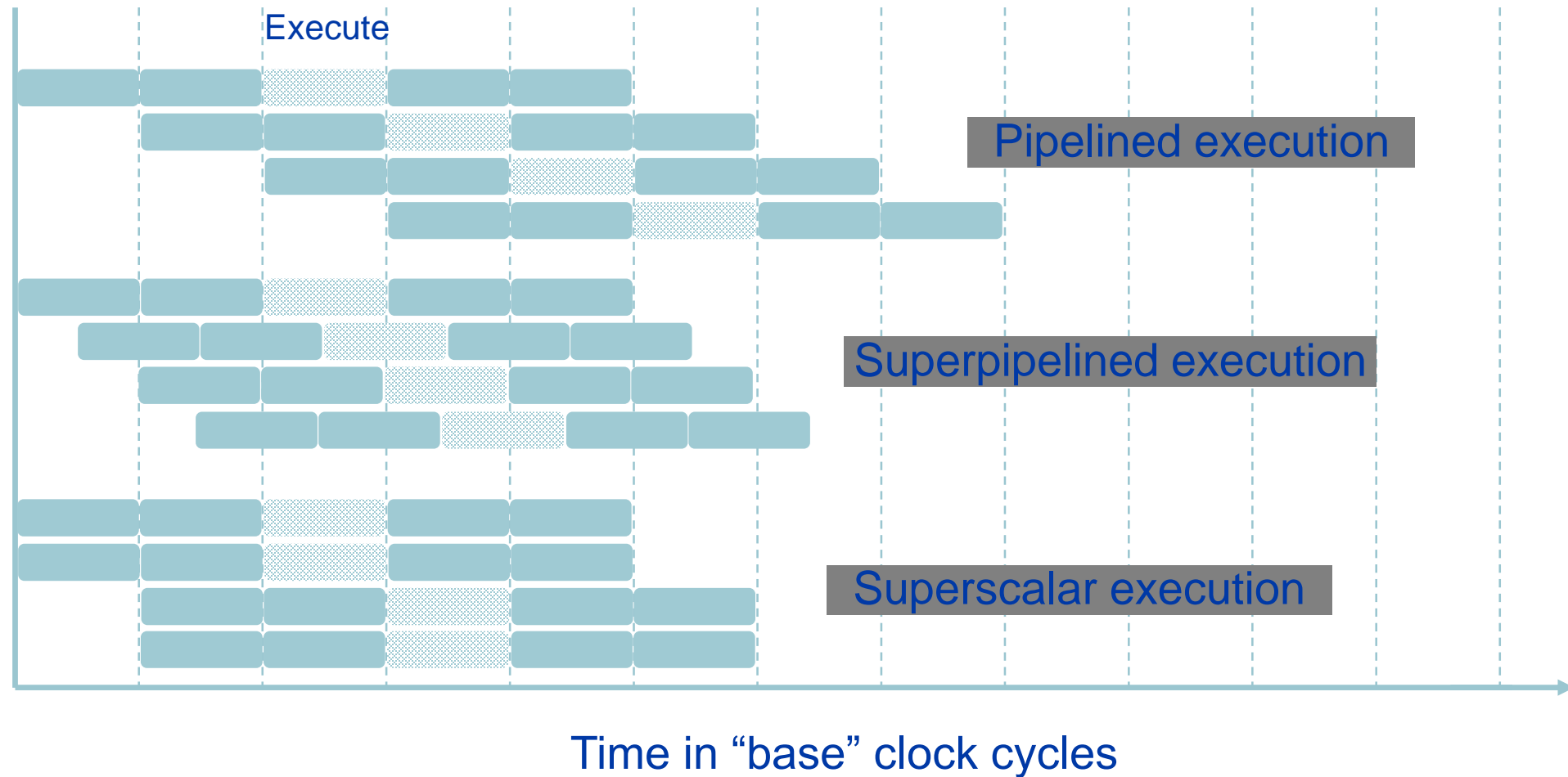
A Superscalar Processor

P instructions are processed in each pipeline stage.

$$IPC \leq P$$

$$\text{Clock Period} = T/S + C$$

Superpipelined and Superscalar Processors



Superpipelined and Superscalar Processors

- If we ignore implementation issues, a superpipelined machine of degree M and a superscalar machine of degree P should have roughly the same performance.
- In either case, we must find $(M \text{ or } P)$ independent instructions from the program that can execute in parallel in each clock cycle. We could use software or hardware techniques to do this.

Superpipelined vs. Superscalar Processors

- In practice, it has proved better to produce superscalar processors, often with deep pipelines, rather than purely superpipelined processors:
 - Practical limits to clock frequency
 - Some operations or modules are difficult to pipeline
 - The need to balance logic in pipeline stages

An Opportunity...

- Consider the example

add x6, x7, x5

add x4, x0, x2

- We can execute in parallel!

- How about this?

sub **x6**, x7, x5

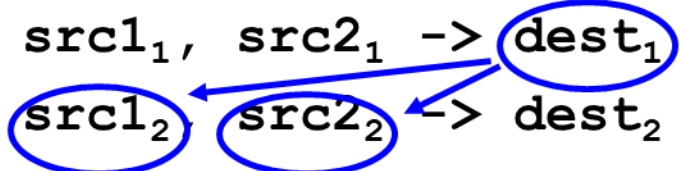
sub x4, **x6**, x0

- In this case, dependences prevent parallel execution!

What Checking Is Required?

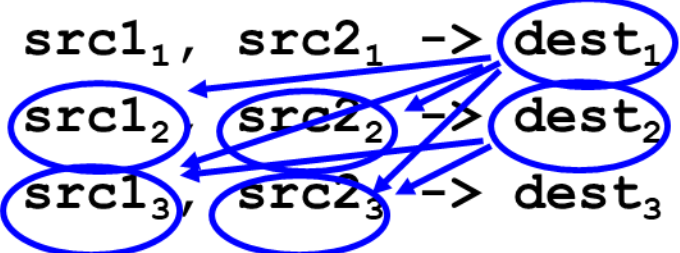
- For two instructions: 2 checks

ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)



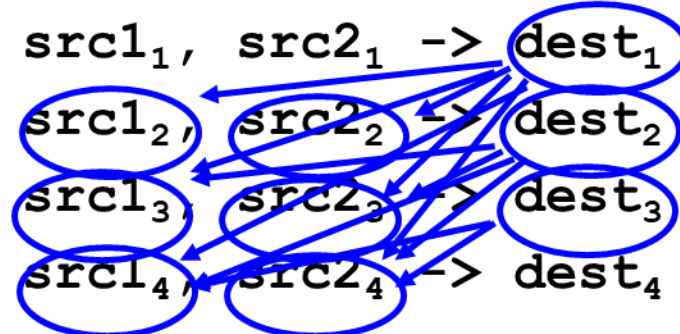
- For three instructions: 6 checks

ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)
ADD src1₃, src2₃ -> dest₃ (4 checks)



- For four instructions: 12 checks

ADD src1₁, src2₁ -> dest₁
ADD src1₂, src2₂ -> dest₂ (2 checks)
ADD src1₃, src2₃ -> dest₃ (4 checks)
ADD src1₄, src2₄ -> dest₄ (6 checks)



- Plus checking for load-to-use stalls from prior n loads

HOW TO BUILD A SUPERSCALAR?

Superscalar

- We could simply fetch two (or more) instructions per clock cycle and, if they are independent, issue them together to different functional units.
- What extra hardware will this processor require?
 - extra logic in decode stage to decode two instructions and check for dependencies
 - register file ports? (extra read and write ports)
 - functional units?
 - additional data forwarding paths?

In-order vs. Out-of-order Execution

■ In-order Execution

- Instructions are fetched, executed and computed in compiler-generated order
- one stalls, they all stall
- instructions are “statically scheduled” by the hardware

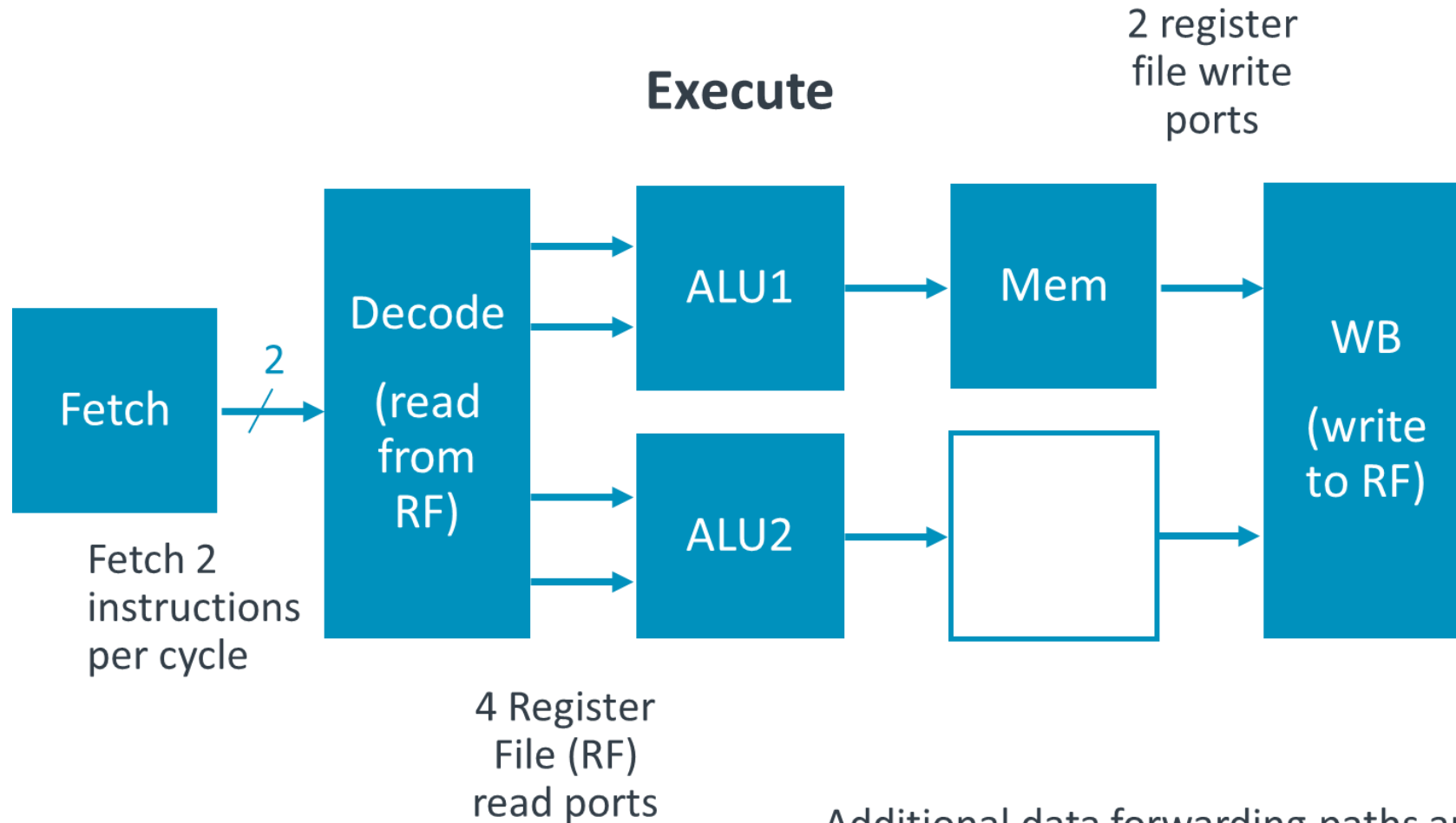
■ Out-of-order (OoO or O3 or O³) Execution

- Instructions are fetched in compiler-generated order
- Instructions are completed in order
- They might be executed in some other order
- instructions behind a stalled instruction can bypass it
- instructions are “dynamically scheduled” by the hardware

Simple In-order Superscalar Processors

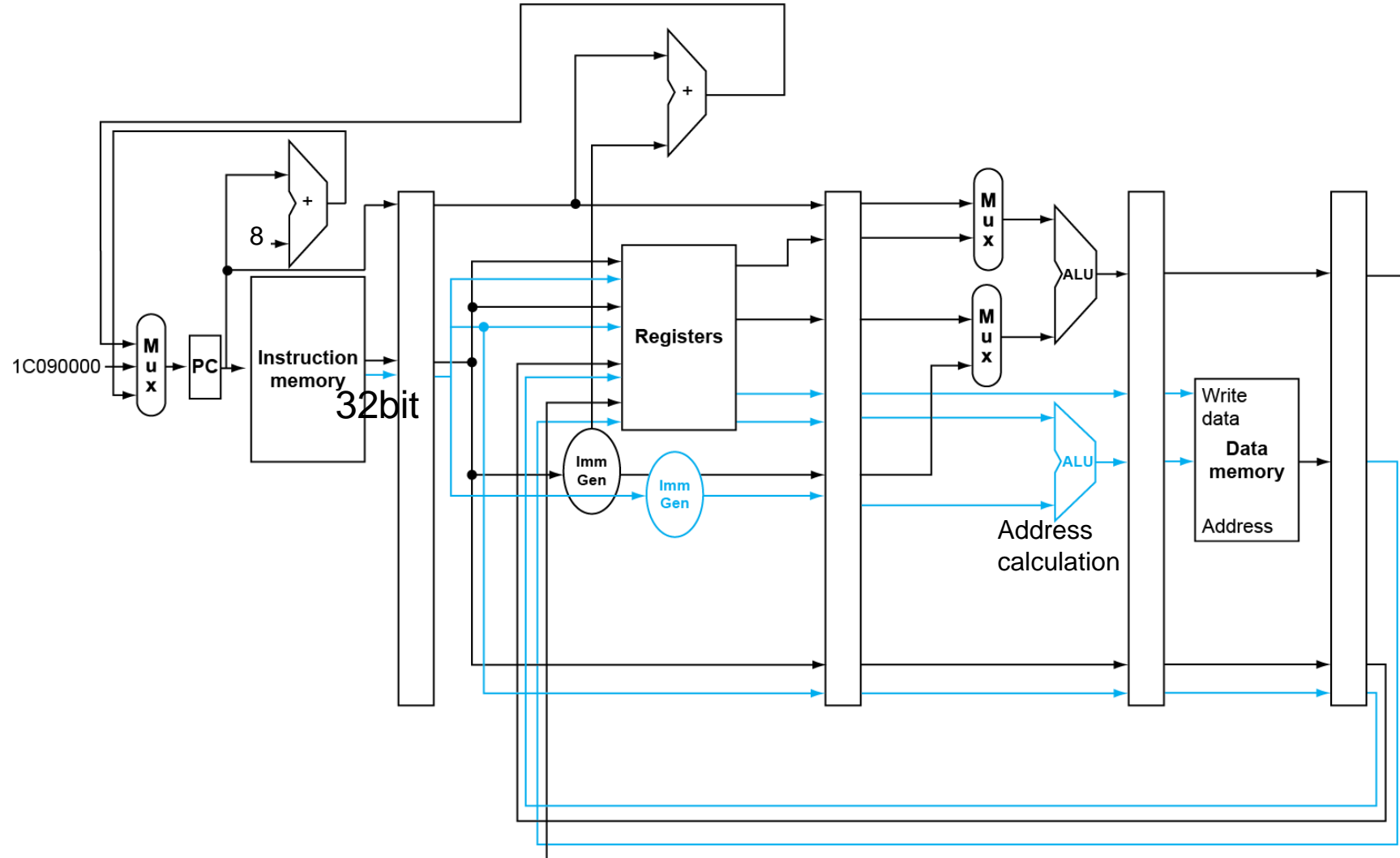
- We can create a simple (**2-way**) superscalar processor with a few changes to our scalar pipeline.
- We will fetch and decode multiple instructions per cycle.
- Instructions are sent to functional units in program order (in-order issue).
- We will issue and execute instructions in parallel if we can.
- If we can't issue two instructions together, we simply issue one and then try to issue the waiting instruction on the next cycle.

Simple In-order Superscalar Processors

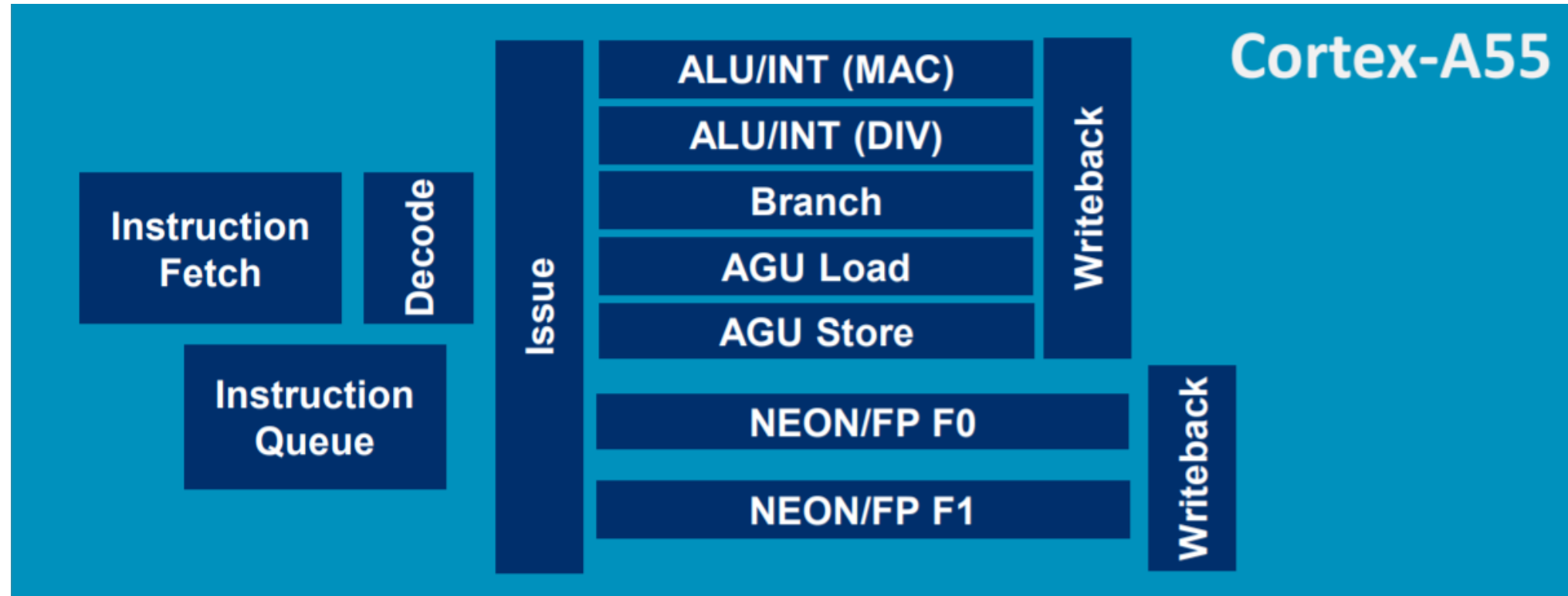


Additional data forwarding paths are also required (not shown here), from and to both ALUs.

RISC-V with Static Dual Issue



A real example: Arm Cortex-A55



2-wide instruction fetch, in-order “dual” instruction issue, 8-stage integer pipeline (Armv8.2-A architecture)

How Much ILP is There?

- The compiler tries to “schedule” code to avoid stalls
 - Even for scalar machines (to fill load-use delay slot)
 - Even harder to schedule multiple-issue (superscalar)
- How much ILP is common?
 - Greatly depends on the application
 - Consider memory copy
 - Unroll loop, lots of independent operations
 - Other programs, less so
- Even given unbounded ILP, superscalar has implementation limits
 - IPC (or CPI) vs clock frequency trade-off
 - Given these challenges, what is reasonable today?
 - ~4 instruction per cycle maximum
 - Intel Broadwell issues up to 8 in the right circumstances, Ryzen up to 6, ARM cores usually issue less

IMPLEMENTATION CHALLENGES

Superscalar Pipeline - Ideal

scalar

```
lw 0(r1) → r2
lw 4(r1) → r3
lw 8(r1) → r4
add r14,r15 → r6
add r12,r13 → r7
add r17,r16 → r8
lw 0(r18) → r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r14,r15 → r6				F	D	X	M	W				
add r12,r13 → r7					F	D	X	M	W			
add r17,r16 → r8						F	D	X	M	W		
lw 0(r18) → r9							F	D	X	M	W	

2-way superscalar

```
lw 0(r1) → r2
lw 4(r1) → r3
lw 8(r1) → r4
add r14,r15 → r6
add r12,r13 → r7
add r17,r16 → r8
lw 0(r18) → r9
```

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3	F	D	X	M	W							
lw 8(r1) → r4		F	D	X	M	W						
add r14,r15 → r6		F	D	X	M	W						
add r12,r13 → r7			F	D	X	M	W					
add r17,r16 → r8			F	D	X	M	W					
lw 0(r18) → r9				F	D	X	M	W				

Superscalar Pipeline - Reality

scalar

lw 0(r1) → r2

lw 4(r1) → r3

lw 8(r1) → r4

add r4, r5 → r6

add r2, r3 → r7

add r7, r6 → r8

lw 4(r8) → r9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3		F	D	X	M	W						
lw 8(r1) → r4			F	D	X	M	W					
add r4, r5 → r6				F	D	d*	X	M	W			
add r2, r3 → r7					F	d*	D	X	M	W		
add r7, r6 → r8							F	D	X	M	W	
lw 4(r8) → r9								F	D	X	M	W

2-way superscalar

lw 0(r1) → r2

lw 4(r1) → r3

lw 8(r1) → r4

add r4, r5 → r6

add r2, r3 → r7

add r7, r6 → r8

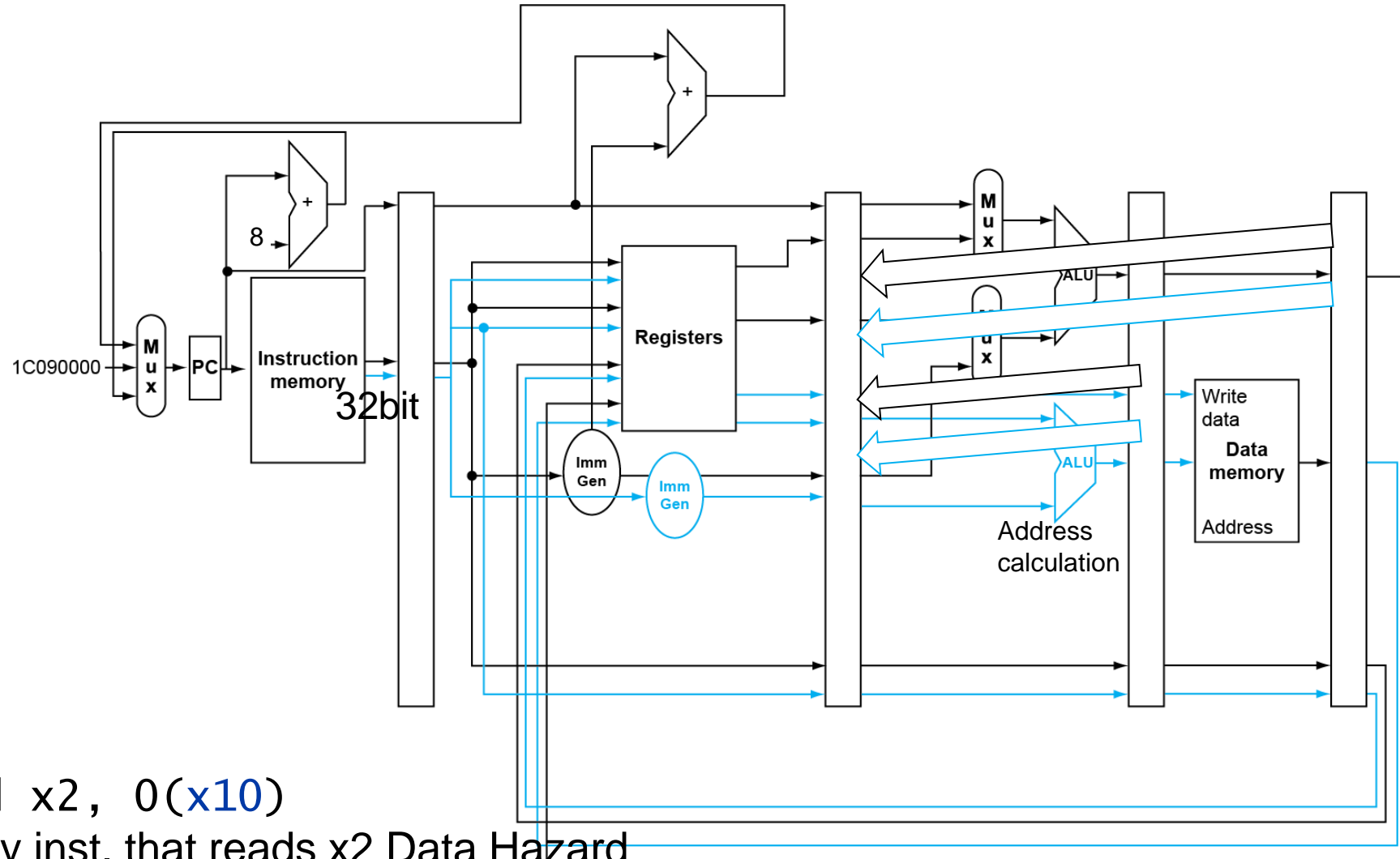
lw 4(r8) → r9

	1	2	3	4	5	6	7	8	9	10	11	12
lw 0(r1) → r2	F	D	X	M	W							
lw 4(r1) → r3	F	D	X	M	W							
lw 8(r1) → r4		F	D	X	M	W						
add r4, r5 → r6		F	D	d*	d*	X	M	W				
add r2, r3 → r7			F	D	d*	X	M	W				
add r7, r6 → r8				F	d*	D	X	M	W			
lw 4(r8) → r9				F	d*	d*	D	X	M	W		

Example: Hazards in the Dual-Issue RISC-V

- More instructions executing in parallel
- 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
 - lw 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

Forwarding paths in Dual-Issue RISC-V



1d x2, 0(x10)

any inst. that reads x2 Data Hazard

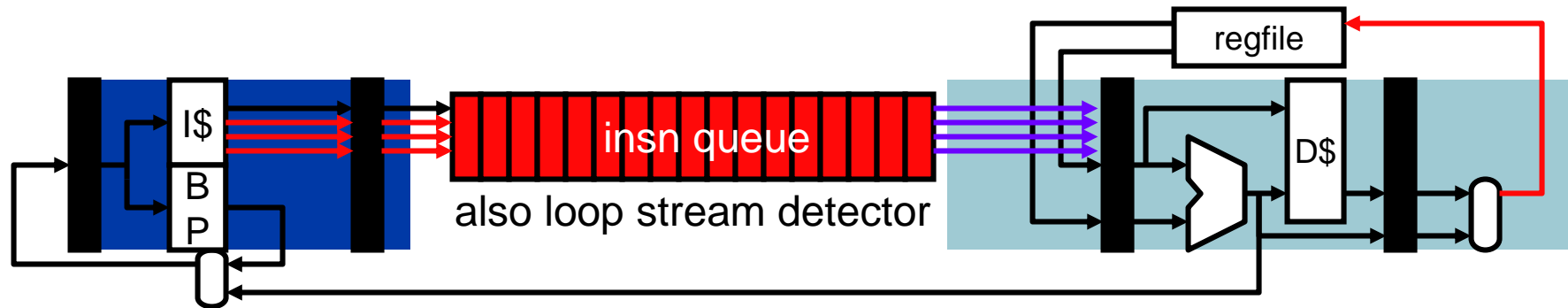
Superscalar Challenges - Front End

- **Superscalar instruction fetch**
 - Modest: fetch multiple instructions per cycle
 - Aggressive: buffer instructions and/or predict multiple branches
- **Superscalar instruction decode**
 - Replicate decoders
- **Superscalar instruction issue**
 - Determine when instructions can proceed in parallel
 - More complex stall logic - order N^2 for N -wide machine
 - Not all combinations of types of instructions possible
- **Superscalar register read**
 - Port for each register read (4-wide superscalar → 8 read “ports”)
 - Each port needs its own set of address and data wires
 - Latency & area $\propto \text{\#ports}^2$

Challenges of Superscalar Fetch

- What is involved in fetching multiple instructions per cycle?
- In same cache block? no problem
 - 64-byte cache block is 16 instructions (~4 bytes per instruction)
 - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
 - Fetch only one instruction that cycle
 - Or, some processors may allow fetching from 2 consecutive blocks
- What about taken branches?
 - How many instructions can be fetched on average?
 - Average number of instructions per taken branch?
 - Assume: 20% branches, 50% taken → ~10 instructions
- Consider a 5-instruction loop with an 4-issue processor
 - Without smarter fetch, ILP is limited to 2.5 (not 4, which is bad)

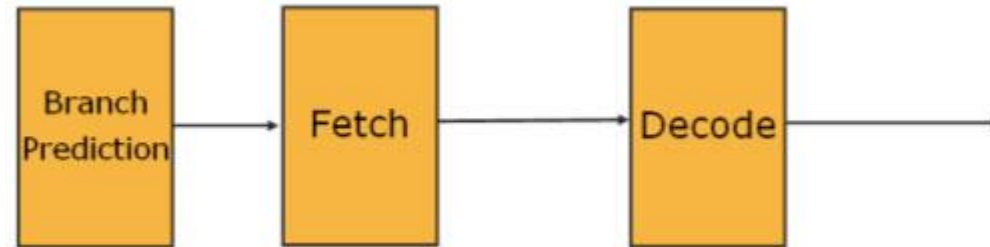
Increasing Superscalar Fetch Rate



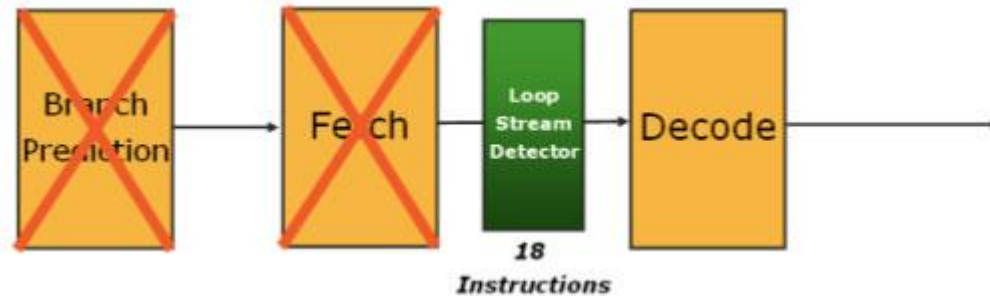
- Option #1: over-fetch and buffer
 - Add a queue between fetch and decode (18 entries in Intel Core2)
 - Compensates for cycles that fetch less than maximum instructions
 - “decouples” the “front end” (fetch) from the “back end” (execute)
- Option #2: “loop stream detector” (Core 2, Core i7)
 - Put entire loop body into a small cache
 - Core2: 18 macro-ops, up to four taken branches
 - Core i7: 28 micro-ops (avoids re-decoding macro-ops!)
 - Any branch mis-prediction requires normal re-fetch
- Other options: next-*next*-block prediction, “trace cache”

Loop Stream Detector

The traditional prediction pipeline



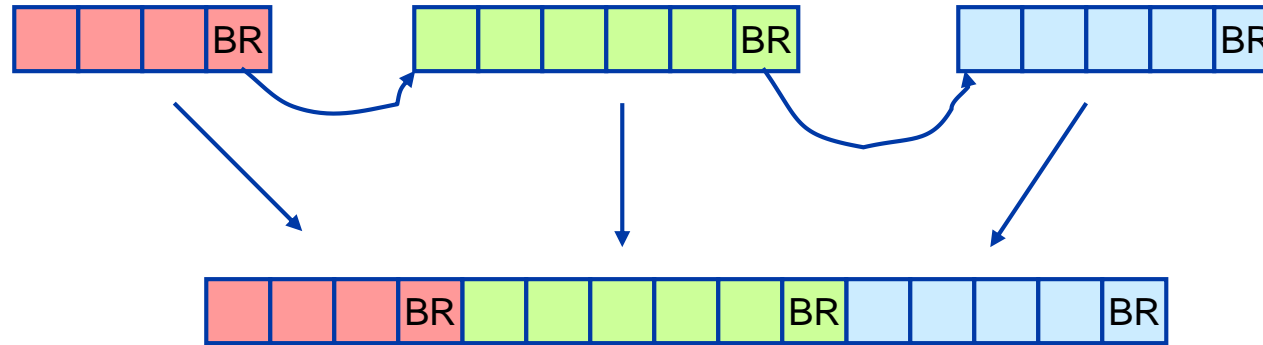
The LSD active on pipeline



- Detect when the CPU was executing a loop in software, stop predicting branches (and potentially incorrectly predicting the last branch of the loop) and simply stream instructions out of the LSD.
- Hold up to 18 instructions in the LSD and simply stream them over and over again into the decode engine until the loop was completed or you ran out of instructions in the LSD.

Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line

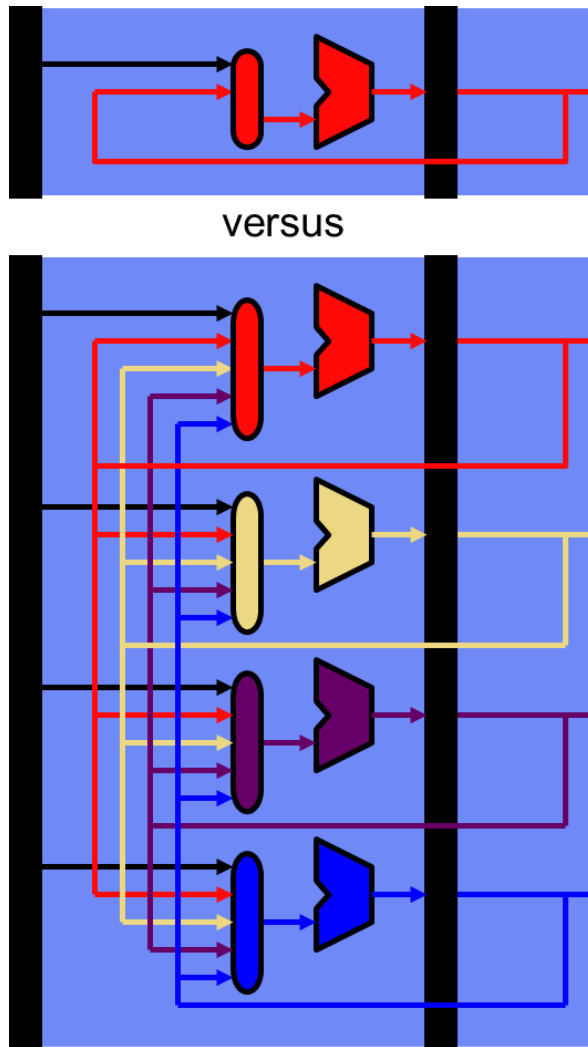


- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next n branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

Superscalar Challenges - Back End

- **Superscalar instruction execution**
 - Replicate arithmetic units (but not all, for example, integer divider)
 - Perhaps multiple cache ports (slower access, higher energy)
 - Only for 4-wide or larger (why? only ~35% are load/store insn)
- **Superscalar bypass paths**
 - More possible sources for data values
 - Order ($N^2 * P$) for N -wide machine with execute pipeline depth P
- **Superscalar instruction register writeback**
 - One write port per instruction that writes a register
 - Example, 4-wide superscalar → 4 write ports
- **Fundamental challenge:**
 - Amount of ILP (instruction-level parallelism) in the program
 - Compiler must schedule code and extract parallelism

Superscalar Bypass

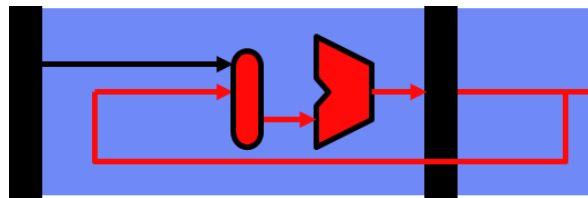


A quick quiz

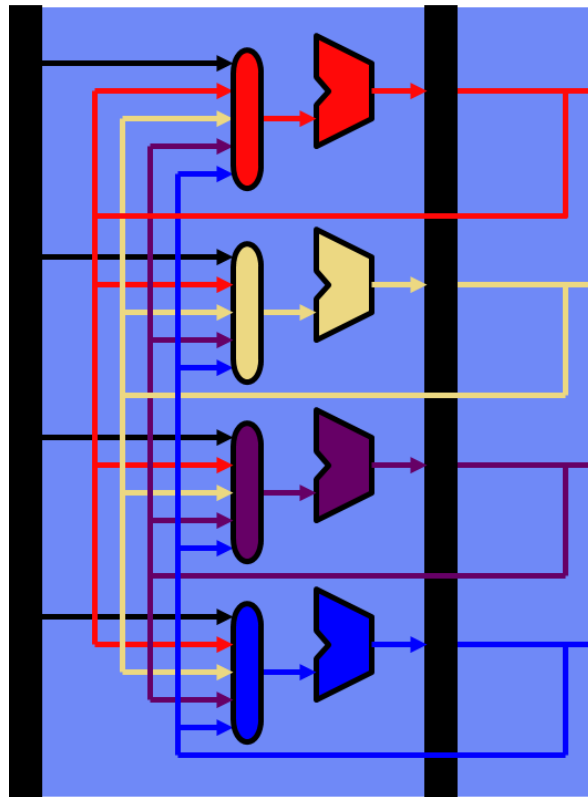
For a N-way superscalar shown on the left, if we only implement one type of bypass path, how many inputs do muxes have at each ALU input?

- A. $2N$
- B. N^2
- C. N
- D. $N+1$
- E. 2

Superscalar Bypass



versus



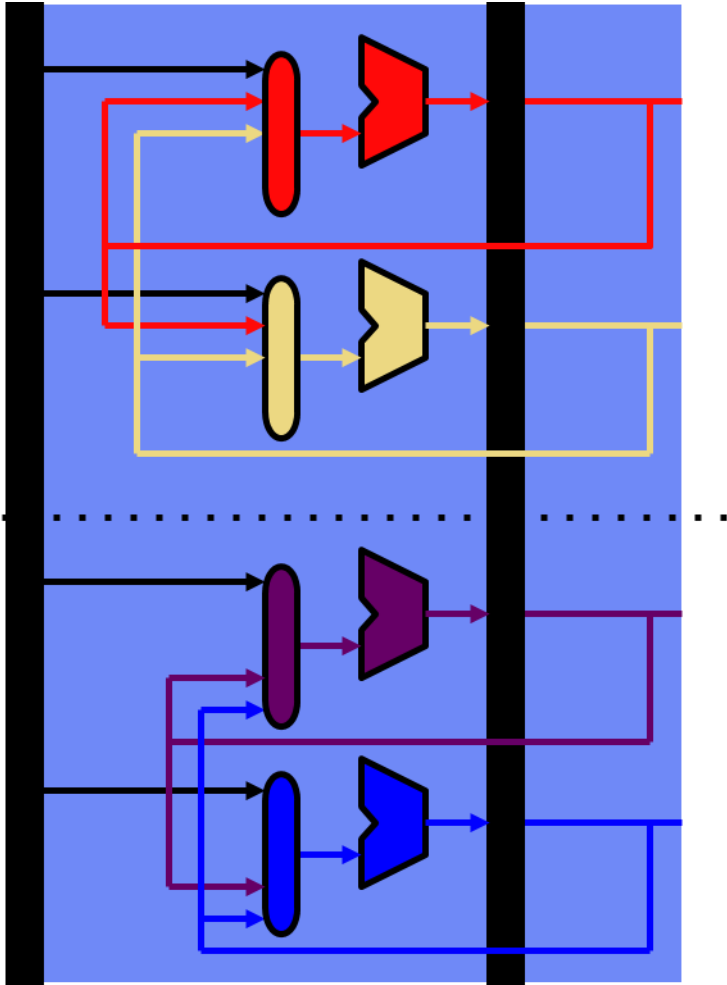
■ N^2 bypass network

- $N+1$ input muxes at each ALU input
 - N^2 point-to-point connections
 - Routing lengthens wires
 - Heavy capacitive load
- And this is just one bypass stage (from beginning of MEM)!
 - There is also other bypassing (from beginning of WB)
 - Even more for deeper pipelines
 - One of the big problems of superscalar
 - Why? On the critical path of single-cycle “bypass & execute” loop

Not All N^2 Created Equal

- N^2 bypass vs. N^2 stall logic & dependence cross-check
 - Which is the bigger problem?
- N^2 bypass ... by far
 - 64-bit quantities (vs. 5-bit)
 - Multiple levels (from MEM, WB) of bypass (vs. 1 level of stall logic)
 - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest N^2 problem
 - Regfile is also an N^2 problem (think latency where N is #ports)
 - And also more serious than cross-check

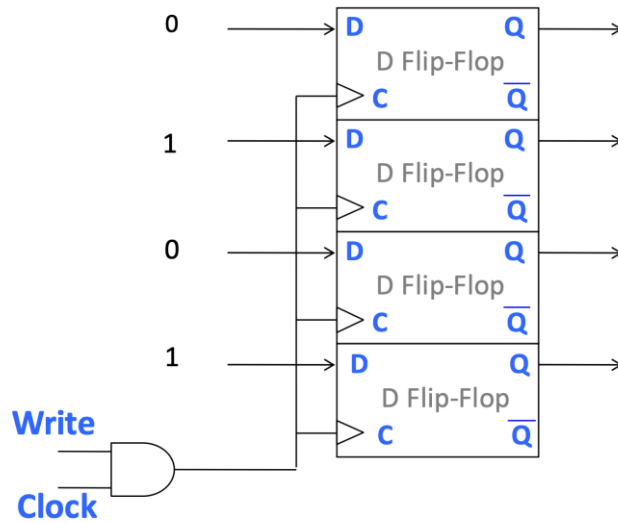
Mitigating N² Bypass & Register File



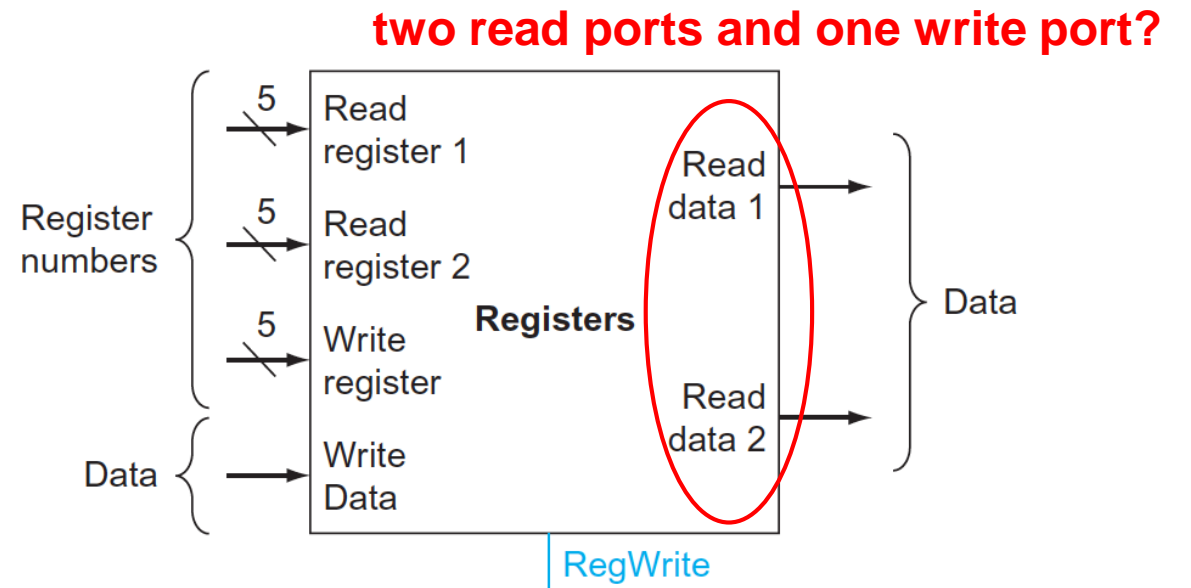
- **Clustering**: mitigates N² bypass
 - Group ALUs into **K** clusters
 - Full bypassing within a cluster
 - Limited bypassing between clusters
 - With 1 or 2 cycle delay
 - Can hurt IPC, but faster clock
 - (N/K) + 1 inputs at each mux
 - (N/K)² bypass paths in each cluster
- **Steering**: key to performance
 - Steer dependent insns to same cluster
- **Cluster register file**, too
 - Replicate a register file per cluster
 - All register writes update all replicas
 - Fewer read ports; only for cluster

RISC-V RegFile (RF)

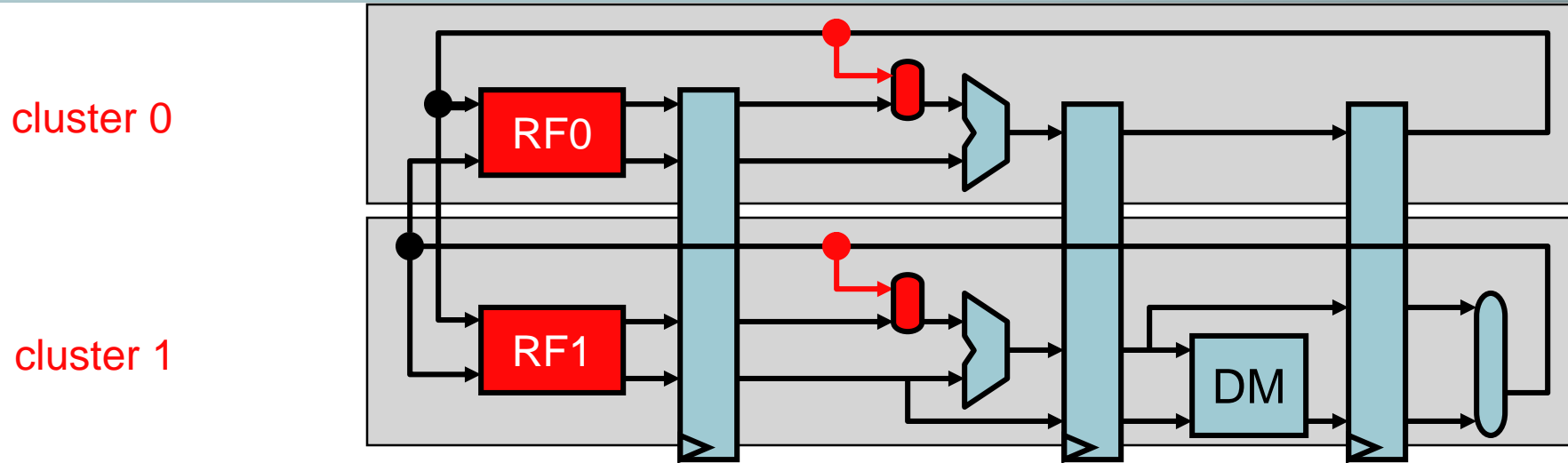
- Collection of registers in which any register can be read or written by specifying the number of the register in the file. The register file contains the register state of the computer.
- A register is a group of flip-flops used to store a binary word



A 4 bit register file



Mitigating N^2 RegFile with Clustering

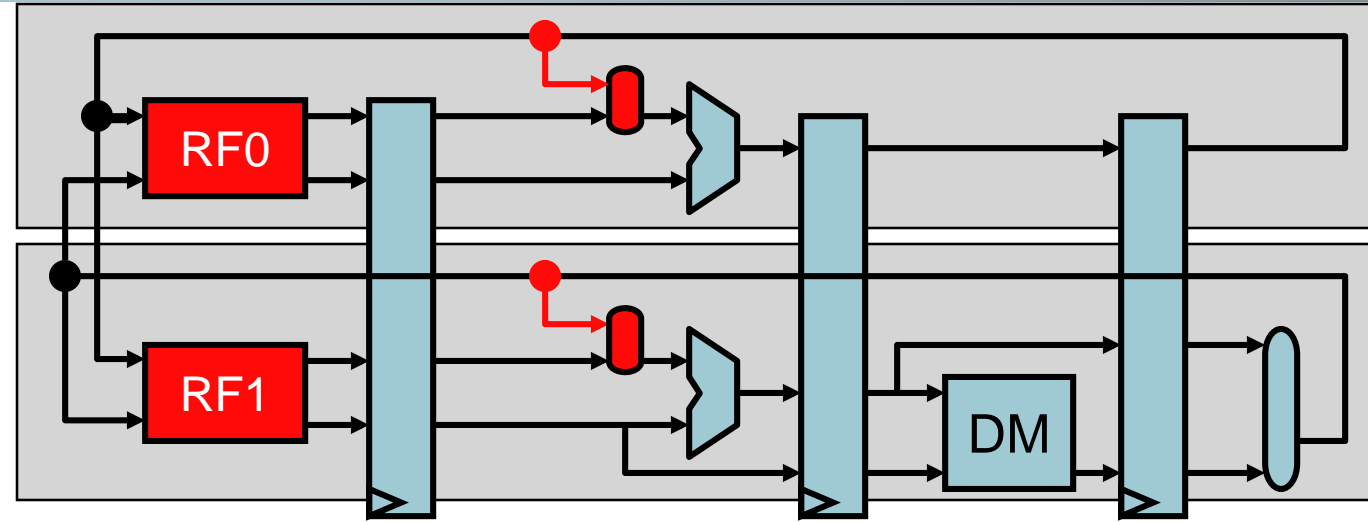


- **Clustering**: split N -wide execution pipeline into K clusters
 - With centralized register file, $2N$ read ports and N write ports
- **Clustered register file**: extend clustering to register file
 - Replicate the register file (one replica per cluster)
 - Register file supplies register operands to just its cluster
 - All register writes go to all register files (keep them in sync)
 - Advantage: fewer read ports per register!

Quiz: Advantage of Clustering

cluster 0

cluster 1



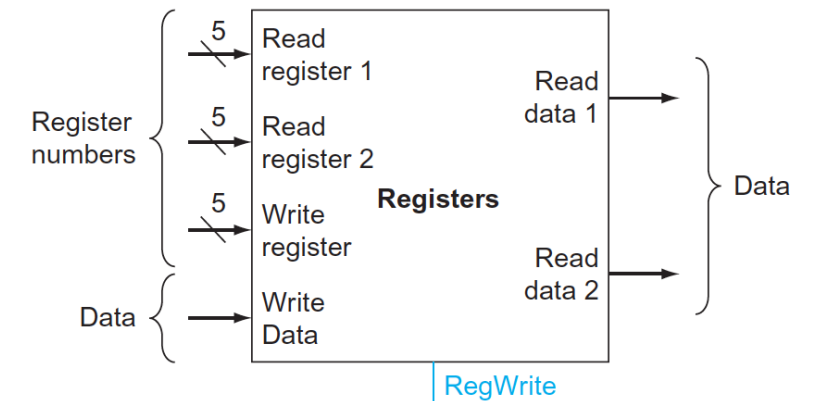
Q: K register files, each with ?? read ports and ?? write ports.

A. $2N/K$, N

B. $2N$, $N-1$

C. N/K , $2N/K$

D. $N/K+1$, $N+1$



Trends in Single-Processor Multiple Issue

	486	Pentium	PentiumII	Pentium4	Itanium	ItaniumII	Core2
Year	1989	1993	1998	2001	2002	2004	2006
Width	1	2	3	3	3	6	4

- Issue width has saturated at 4-6 for high-performance cores
 - Canceled Alpha 21464 was 8-way issue
 - Not enough ILP to justify going to wider issue
 - Hardware or compiler *scheduling* needed to exploit 4-6 effectively
 - More on this in the next topic
- For high-performance ***per watt*** cores (e.g., smart phones)
 - Typically 2-wide superscalar (but increasing each generation)

Superscalar (In-Order) Summary

- Superpipelined and Superscalar Processors
- Multiple issue
 - Exploits insn level parallelism (ILP) beyond pipelining
 - Improves IPC, but perhaps at some clock & energy penalty
 - 4-6 way issue is about the peak issue width currently justifiable
 - Low-power implementations today typically 2-wide superscalar
- Problem spots
 - N^2 bypass & register file → clustering
 - Fetch + branch prediction → buffering, loop streaming, trace cache

Where are we Heading?

- T4: Advanced Processors II

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- HW#1 is upcoming
- Reading Materials
 - Ch. 3.7
 - Ch. Appendix C.2