

ECE 4750 Computer Architecture, Fall 2022

Topic 6: Advanced Processors Superscalar Execution

School of Electrical and Computer Engineering
Cornell University

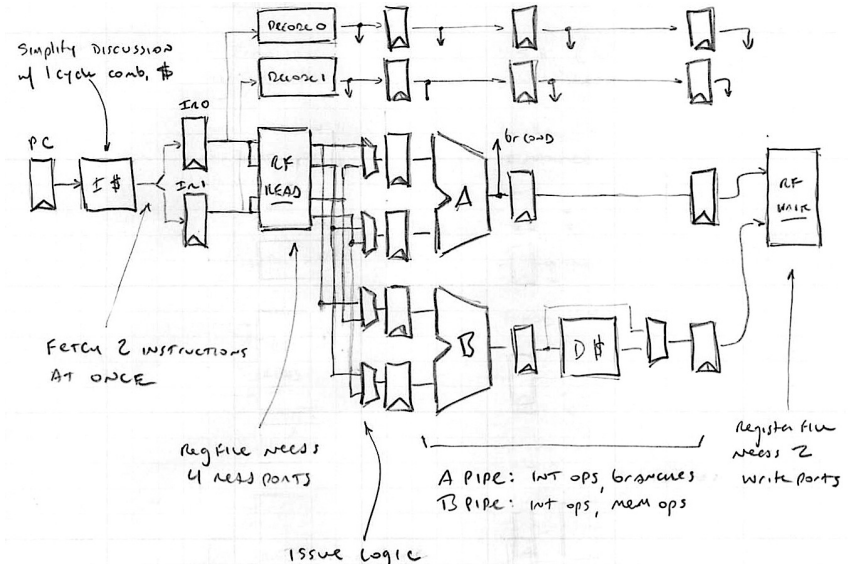
revision: 2022-10-31-08-29

1	In-Order Dual-Issue Superscalar TinyRV1 Processor	2
2	Superscalar Pipeline Hazards	4
2.1.	RAW Hazards	4
2.2.	Control Hazards	6
2.3.	Structural Hazards	10
2.4.	WAW and WAR Name Hazards	10
3	Analyzing Performance of Superscalar Processors	11

Copyright © 2022 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 4750 Computer Architecture. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

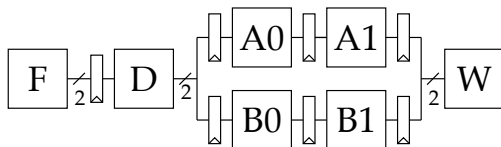
1. In-Order Dual-Issue Superscalar TinyRV1 Processor

- Processors studied so far are fundamentally limited to $CPI \geq 1$
- Superscalar processors enable $CPI < 1$ (i.e., $IPC > 1$) by executing multiple instructions in parallel
- Can have both in-order and out-of-order superscalar processors, but we will start by exploring in-order



- Continue to assume combinational memories
- F Stage** : fetch two instructions at once
- D Stage** : 4 read ports, decode 2 inst, "issue" inst to correct pipe
- X/M Stage** : separate into A and B pipes (see next page)
- W Stage** : 2 write ports

More abstract way to illustrate same dual-issue superscalar pipeline



Different instructions use the A-pipe and/or the B-pipe

	add	addi	mul	lw	sw	jal	jr	bne
A-Pipe	✓	✓	✓			✓	✓	✓
B-Pipe	✓	✓		✓	✓	✓	✓	

Example pipeline diagram for dual-issue superscalar processor

addi x1, x2, 1																		
addi x3, x4, 1																		
addi x5, x6, 1																		
mul x7, x8, x9																		
mul x10, x11, x12																		
addi x13, x14, 1																		

- Multiple instructions in stages F, D, W allowed because superscalar processor has duplicated hardware to avoid structural hazards
- **Fetch Block** – group of instructions fetched as unit
- **Swizzle** – instructions “swapped” from natural fetch position to appropriate execution pipe

Revisit previous assembly sequence with full bypassing

addi x1, x2, 1														
addi x3, x4, 1														
add x5, x1, x3														
addi x6, x5, 1														
addi x7, x8, 1														
addi x9, x8, 1														

Activity: Draw a pipeline diagram for following instruction sequence. Include all microarchitectural dependency arrows.

addi x1, x2, 1														
lw x3, 0(x4)														
lw x5, 0(x3)														
addi x6, x7, 1														
addi x8, x5, 1														
addi x9, x8, 1														

2.2. Control Hazards

Consider following two static instruction sequences.

<pre> 1 0x1000 addi x1, x2, 1 2 0x1004 jal x0, foo 3 ... 4 foo: 5 0x2000 addi x3, x4, 1 6 0x2004 addi x5, x6, 1 </pre>	<pre> 1 # assume R[x1] != R[x2] 2 0x1000 bne x1, x2, foo 3 ... 4 foo: 5 0x2000 addi x3, x4, 1 6 0x2004 addi x5, x6, 1 </pre>
--	--

Pipeline diagram for left sequence. Jumps are resolved in D stage.

Pipeline diagram for right sequence. Branches are resolved in A0 stage.

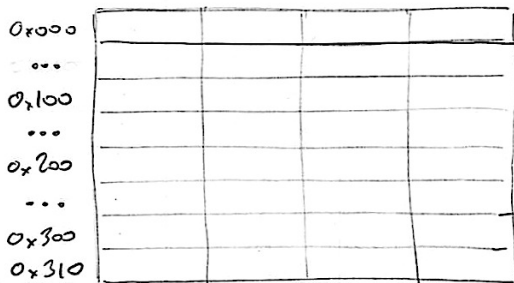
Unaligned fetch blocks

Consider the following static instruction sequence

```

1 0x000 opA
2 0x004 opB
3 0x008 opC
4 0x00c jal x0, 0x100
5 ...
6 0x100 opD
7 0x104 jal x0, 0x204
8 ...
9 0x204 opE
10 0x208 jal x0, 0x30c
11 ...
12 0x30c opF
13 0x310 opG
14 0x314 opH
  
```

Layout of fetch blocks in instruction cache.
Numbers indicate which instructions belong to which fetch block.



op A	F	D	A0	A1	W
op B	F	D	B0	B1	W
op C	F	D	B0	B0	W
J	F	D	A0	A1	W
op D	F	D	B0	B1	W
J	F	D	A0	A1	W
op E	F	D	B0	B1	W
J	F	D	A0	A1	W
op F	F	D	A0	A1	W
op G	F	D	B0	B1	W
op H	F	D	A0	A1	W

- Unaligned fetch blocks within a cache line are challenging
- Unaligned fetch blocks across cache lines are very challenging

Aligned fetch blocks

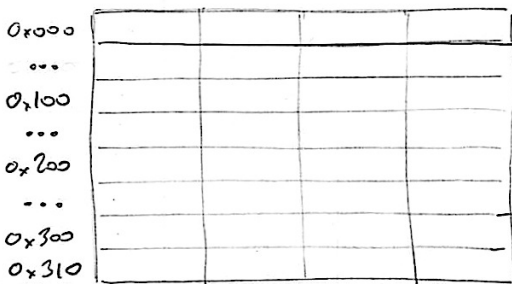
Only fetch aligned fetch blocks, possibly discarding first instruction.
Reconsider the same static instruction sequence

```

1 0x000 opA
2 0x004 opB
3 0x008 opC
4 0x00c jal x0, 0x100
5 ...
6 0x100 opD
7 0x104 jal x0, 0x204
8 ...
9 0x204 opE
10 0x208 jal x0, 0x30c
11 ...
12 0x30c opF
13 0x310 opG
14 0x314 opH

```

Layout of fetch blocks in instruction cache.
Numbers indicate which instructions belong to which fetch block.



0x000	opA	F D A0 A1 W
0x004	opB	F D B0 B1 W
0x008	opC	F D B0 B1 W
0x00c	J	F D A0 A1 W
0x100	opD	F D B0 B1 W
0x104	J	F D A0 A1 W
0x200	?	F - - - -
0x204	opE	F D A0 A1 W
0x208	J	F D A0 A1 W
0x20c	?	F - - - -
0x308	?	F - - - -
0x30c	opF	F D A0 A1 W
0x310	opG	F D A0 A1 W
0x314	opH	F D B0 B1 W

BUBBLES HANDLING ALIGNMENT ISSUES

eventually BACK in "sync"

2.3. Structural Hazards

Structural hazards *are not* possible in the canonical single-issue TinyRV1 pipeline, but structural hazards *are* possible in the canonical dual-issue TinyRV1 pipeline if two instructions in the same fetch block want to use the same pipe.

mul x1, x2, x3																			
mul x4, x5, x6																			
lw x7, 0(x8)																			
sw x9, 0(x10)																			

2.4. WAW and WAR Name Hazards

WAW name hazards *are not* possible in the canonical single-issue TinyRV1 pipeline, but WAW name hazards *are* possible in the canonical dual-issue TinyRV1 pipeline if two instructions in the same fetch block write the same register.

addi x1, x2, 1																			
addi x1, x3, 1																			

WAR name hazards *are not* possible in the canonical single-issue TinyRV1 pipeline. Are WAR name hazards possible in the canonical dual-issue TinyRV1 pipeline?

addi x1, x2, 1																			
addi x2, x3, 1																			

