

Processor Cores

ECE/CS 757 Spring 2007
J. E. Smith

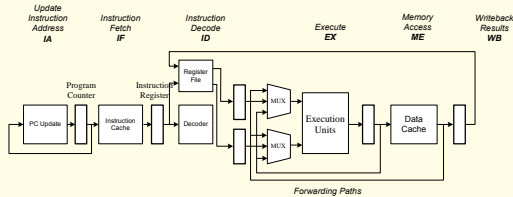
Copyright (C) 2007 by James E. Smith (unless noted otherwise)
All rights reserved. Except for use in ECE/CS 757, no part of these notes may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from the author.

Outline

- Conventional Cores (Review)
 - In-order
 - Out-of-order Superscalar
- Historical Multi-threaded Cores
 - 6600, HEP
- In-Order Multi-threaded Cores
- Out-of-Order Multi-threaded Cores
 - Resource Sharing
 - Thread Scheduling
- Case Studies
 - IBM RS64 IV
 - SUN Niagara
 - IBM Power5

In-Order Pipelines

- As used in early RISCs
- Re-merging in throughput-oriented servers
- Complexity in stall/forwarding logic

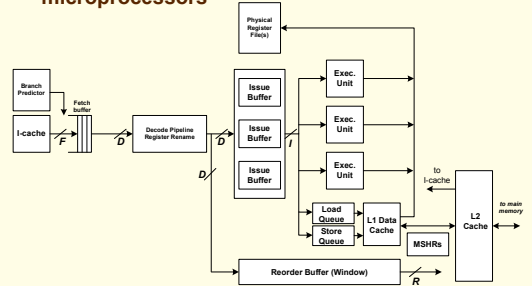


Stall/Forward Example

	1	2	3	4	5	6	7	8	9	10
r1 ← r2 + r3	IA	IF	ID	EX	ME	WB				
r3 ← mem[r1 + 8]		IA	IF	ID	EX	ME	WB			
r5 ← r3 - r4			IA	IF	ID		EX	ME	WB	

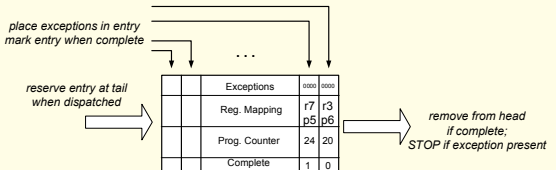
Superscalar Processors

- Widely used today in high performance microprocessors



Superscalar Processors

- Issue instructions out-of-order when dependences are resolved
- Reorder buffer keeps track of architected order



Example

- Instruction fetch and branch prediction converts static program into dynamic instruction stream

<p>Static Program:</p> <pre> loop: r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 </pre>	<p>Dynamic Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 </pre>
---	---

(c) 2007 J. E. Smith

7

Register Renaming

- More physical registers than logical registers
- Physical register names act as tags
 - Mapping changes over time
- Turns instruction stream into "single assignment" form
 - avoids WAW and WAR hazards
- For each instruction
 - Read source register mappings from table
 - Acquire new result physical register from free pool
 - Modify map table for new result register mapping
- Example
 - Assume 8 logical, 24 physical registers (to simplify free pool management in example fewer physical regs. can be used in practice)

(c) 2007 J. E. Smith

8

Example

<p>Fetch Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 </pre>	<p>Register MAP</p> <pre> r1 p3 r2 p4 r3 p6 r4 p1 r5 p2 r6 p7 r7 p5 </pre>	<p>Renamed Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 </pre>
--	---	--

Free Pool

```

p8,p9,p10,p11,p12,p13,
p14,p15,p16,p17,p18,
P19,p20,p21,p22,p23,p24
    
```

(c) 2007 J. E. Smith

9

Example

<p>Fetch Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 </pre>	<p>Register MAP</p> <pre> r1 p3 r2 p4 r3 p6 r4 p1 r5 p2 r6 p7 r7 p5 </pre>	<p>Renamed Stream</p> <pre> r3 <- mem(p1+p4) </pre>
--	---	---

Free Pool

```

p8,p9,p10,p11,p12,p13,
p14,p15,p16,p17,p18,
P19,p20,p21,p22,p23,p24
    
```

(c) 2007 J. E. Smith

10

Example

<p>Fetch Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 </pre>	<p>Register MAP</p> <pre> r1 p3 r2 p4 r3 p6 r4 p1 r5 p2 r6 p7 r7 p5 </pre>	<p>Renamed Stream</p> <pre> p8 <- mem(p1+p4) </pre>
--	---	---

Free Pool

```

p8,p9,p10,p11,p12,p13,
p14,p15,p16,p17,p18,
P19,p20,p21,p22,p23,p24
    
```

(c) 2007 J. E. Smith

11

Example

<p>Fetch Stream</p> <pre> r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 P <- loop; r1!=0 r3 <- mem(r4+r2) r7 <- mem(r5+r2) r7 <- r7 * r3 r1 <- r1 - 1 mem(r6+r2) <- r7 r2 <- r2 + 8 </pre>	<p>Register MAP</p> <pre> r1 p3 r2 p4 r3 p8 r4 p1 r5 p2 r6 p7 r7 p5 </pre>	<p>Renamed Stream</p> <pre> p8 <- mem(p1+p4) </pre>
--	---	---

Free Pool

```

p9,p10,p11,p12,p13,
p14,p15,p16,p17,p18,
P19,p20,p21,p22,p23,p24
    
```

(c) 2007 J. E. Smith

12

Example

Fetches Stream

```

r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8

```

Register MAP

```

r1 p3
r2 p4
r3 p8
r4 p1
r5 p2
r6 p7
r7 p9

```

Free Pool

```

p10,p11,p12,p13, p14,p15,
p16,p17,p18,P19,p20,p21,p2
2,p23,p24

```

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)

```

(c) 2007 J. E. Smith

13

Example

Fetches Stream

```

r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8

```

Register MAP

```

r1 p3
r2 p4
r3 p8
r4 p1
r5 p2
r6 p7
r7 p10

```

Free Pool

```

p11,p12,p13, p14,p15,
p16,p17,p18,P19,p20,p21,
p22, p23,p24

```

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8

```

(c) 2007 J. E. Smith

14

Example

Fetches Stream

```

r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8
P <- loop; r11=0
r3 <- mem(r4+r2)
r7 <- mem(r5+r2)
r7 <- r7 * r3
r1 <- r1 - 1
mem(r6+r2) <- r7
r2 <- r2 + 8

```

Register MAP

```

r1 p21
r2 p22
r3 p18
r4 p1
r5 p2
r6 p7
r7 p20

```

Free Pool

```

p23,p24

```

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11=0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16=0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8

```

(c) 2007 J. E. Smith

15

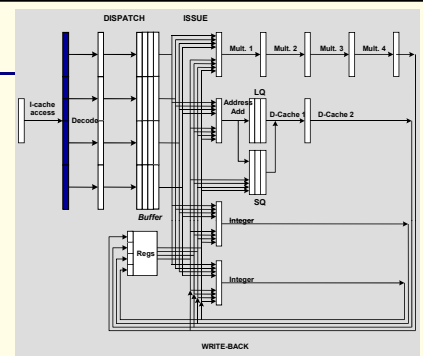
Example

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11=0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16=0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8

```



(c) 2007 J. E. Smith

16

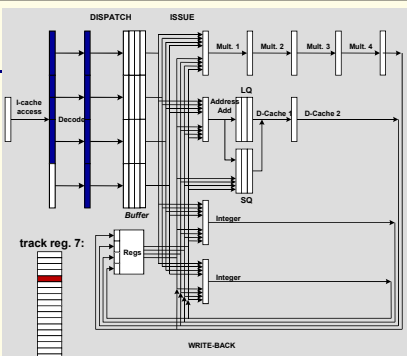
Example

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11=0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16=0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8

```



(c) 2007 J. E. Smith

17

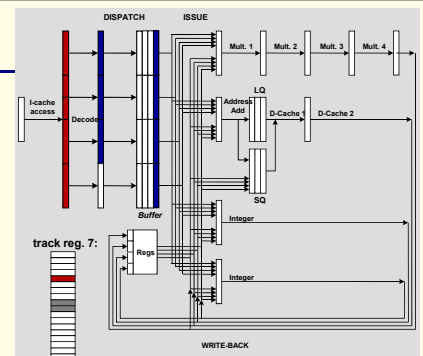
Example

Renamed Stream

```

p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10 <- p9 * p8
p11 <- p3 - 1
mem(p7+p4) <- p10
p12 <- p4 + 8
P <- loop; p11=0
p13 <- mem(p1+p12)
p14 <- mem(p2+p12)
p15 <- p14 * p13
p16 <- p11 - 1
mem(p7+p12) <- p15
p17 <- p12 + 8
P <- loop; p16=0
p18 <- mem(p1+p17)
p19 <- mem(p2+p17)
p20 <- p19 * p18
p21 <- p16 - 1
mem(p7+p17) <- p20
p22 <- p17 + 8

```



(c) 2007 J. E. Smith

18

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 19

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 20

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 21

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 22

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 23

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 24

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 25

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 26

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 27

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 28

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 29

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8
  
```

(c) 2007 J. E. Smith 30

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8

```

(c) 2007 J. E. Smith 31

Example

```

Renamed Stream
p8 <- mem(p1+p4)
p9 <- mem(p2+p4)
p10<- p9 * p8
p11<- p3 - 1
mem(p7+p4)<- p10
p12<- p4 + 8
P <- loop; p11=0
p13<- mem(p1+p12)
p14<- mem(p2+p12)
p15<- p14 * p13
p16<- p11 - 1
mem(p7+p12)<- p15
p17<- p12 + 8
P <- loop; p16=0
p18<- mem(p1+p17)
p19<- mem(p2+p17)
p20<- p19 * p18
p21<- p16 - 1
mem(p7+p17)<- p20
p22<- p17 + 8

```

(c) 2007 J. E. Smith 32

Example

Renamed Stream	dispatch	issue	complete
p8 <- mem(p1+p4)	0	1	4
p9 <- mem(p2+p4)	0	2	5
p10<- p9 * p8	0	5	10
p11<- p3 - 1	0	1	2
mem(p7+p4)<-p10	1	3	12
p12<- p4 + 8	1	2	3
P <- loop; p11=0	1	2	3
p13<- mem(p1+p12)	2	4	7
p14<- mem(p2+p12)	2	5	8
p15<- p14 * p13	2	8	13
p16<- p11 - 1	2	3	4
mem(p7+p12)<- p15	3	6	15
P <- loop; p16=0	3	4	5
p18<- mem(p1+p17)	4	7	10
p19<- mem(p2+p17)	4	8	11
p20<- p19 * p18	4	11	16
p21<- p16 - 1	4	5	6
mem(p7+p17)<- p20	5	9	18
p22<- p17 + 8	5	6	7

Performance:
1 loop iteration per 3 cycles

(c) 2007 J. E. Smith 33

Bottlenecks

- Initially
 - 7 instructions per 3 clocks => 2.3 IPC
 - Bottleneck is cache ports (1)
- Add cache port
 - 7 instructions per 2 cycles => 3.5 IPC
 - Bottleneck is I-fetch
- Increase fetch width (F) to 8
 - 4 instructions per cycle
 - (But I didn't check this completely)

(c) 2007 J. E. Smith 34

Exceptions and Commit

- Exceptions are reported to ROB entry as they are detected
- When an instruction completes execution, this is marked in ROB
- Instructions are removed from ROB in order only after complete

place exceptions in entry
mark entry when complete

reserve entry at tail
when dispatched

remove from head
if complete;
STOP if exception present

(c) 2007 J. E. Smith 35

Exceptions and Commit

- If no exceptions
 - Previous register map discarded => commit of register state
 - Signal store queue to commit store to memory
- If exception
 - Stop committing
 - Restore register mappings from tail to head of ROB
 - Restore PC of excepting instruction

place exceptions in entry
mark entry when complete

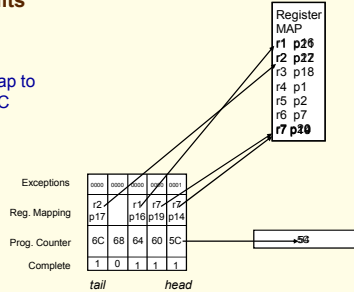
reserve entry at tail
when dispatched

remove from head
if complete;
STOP if exception present

(c) 2007 J. E. Smith 36

Example

- Final load in previous example page faults
 - at PC=5C
- Use ROB to
 - Restore register map to state just prior to 5C
 - Restore PC to 5C



(c) 2007 J. E. Smith

37

Load/Store Buffering

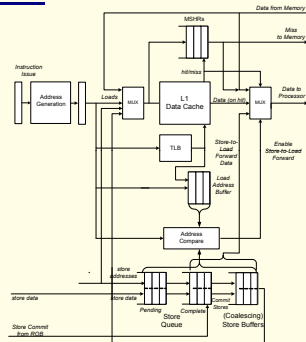
- Important component with respect to Multiprocessor Design
 - For both memory ordering problems and solutions
- Problem: memory address dependences not resolvable until after issue
 - Addresses must first be computed
- Performance advantages if:
 - Loads can pass stores
 - Pending stores forward data to loads

(c) 2007 J. E. Smith

38

General Load/Store Implementation

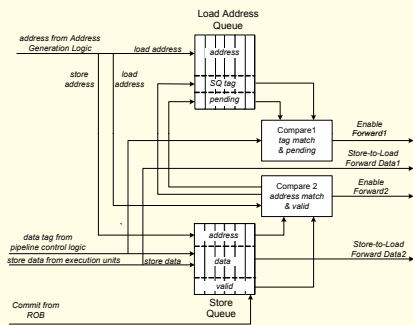
- Store assigned Store Queue entry at dispatch
- Loads and Stores issue in order
- Stores issue when address registers ready
 - Data forwarded later (Store completes)
 - Stores commit when at head of ROB
- Load addresses compared with pending stores
 - Match when store data available => forward
 - Match when data unavailable => wait for data, then forward
- Coalescing store buffers reduce cache bandwidth demand



(c) 2007 J. E. Smith

39

Forwarding: More Detail

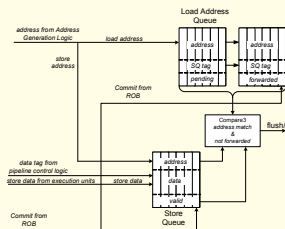


(c) 2007 J. E. Smith

40

More aggressive implementation

- Issue loads and stores out of order
 - Before addresses are known
- Additional buffering/checking
- Detects address match when store addresses are known
 - Save loads in completed load buffer
- On address match, flush and restart using ROB mechanism
- Add-on optimizations (not shown) predict conflicts and defer load issue

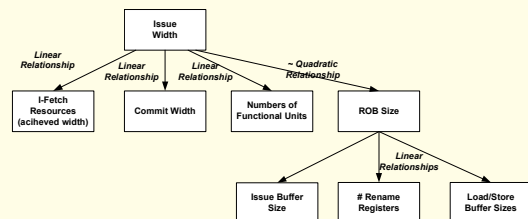


(c) 2007 J. E. Smith

41

Balanced Superscalar Processors

- Relationships between bandwidths and buffer sizes for balanced design
- Most bandwidths linear with respect to issue width
- Most storage quadratic (or more) with respect to issue width



(c) 2007 J. E. Smith

42

Balance Relationships

- Empirically derived relationship between Issue width (I) and Window size (W) – based on critical dependence paths
 $W^{1/2} \leq I \leq W^{1/2}$
- Issue buffer size $\approx W/3$

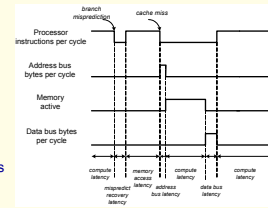
Processor	Reorder Buffer Size	Issue Buffer Size	Issue Width	$\log_2(\text{ROB}) - \log_2(\text{Issue Width})$	Issue Buffer Size - ROB Size
Intel Core	96	32	4	3.3	.3
IBM Power4	100	36	5	2.9	.4
MIPS R10000	64	20	4	3.0	.3
Intel PentiumPro	40	20	3	3.4	.5
Alpha 21264	80	20	4	3.2	.25
AMD Opteron	72	24	4	3.1	.3
HP PA-8000	56		4	2.9	
Intel Pentium 4	120		3	4.4	

(c) 2007 J. E. Smith

43

Multithreaded Processors

- Motivation: many resources (including internal processor resources) are used intermittently
- Let threads (or processes) share a single hardware processor
- Replicate hardware that holds architected state
 - program counter, logical registers
- Share or partition implementation hardware
 - functional units, caches, physical registers, pipeline hardware
 - implementation hardware may be expanded/enhanced
- Multithreading is implementation, not architecture



(c) 2007 J. E. Smith

44

Historical Multithreaded Processors

- CDC6600 PPs
 - I/O processing
- Denelcor HEP
 - General purpose scientific

(c) 2007 J. E. Smith

45

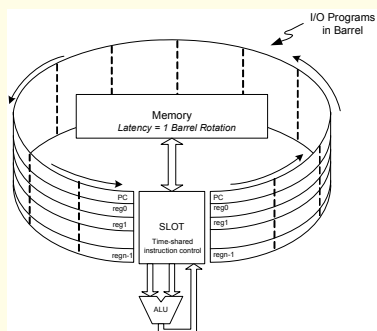
CDC 6600 Peripheral Processors

- Intended to perform OS and I/O functions
- Used "barrel and slot"
 - register state is arranged around a "barrel"
 - one set of ALU and memory hardware accessed through "slot" in barrel
 - slot in barrel rotates one position each cycle
- Could be used as stand-alone "MP"
- Similar method later used in IBM Channels

(c) 2007 J. E. Smith

46

CDC 6600 Peripheral Processors

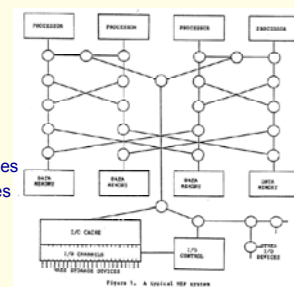


(c) 2007 J. E. Smith

47

Denelcor HEP

- General purpose scientific computer
- Organized as an MP
 - Up to 16 processors
 - Each processor is multithreaded
 - Up to 128 memory modules
 - Up to 4 I/O cache modules
 - Three-input switches and chaotic routing



(c) 2007 J. E. Smith

48

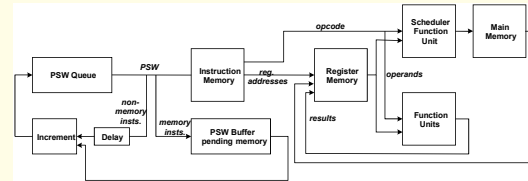
Processor Organization

- ❑ **Multiple contexts (threads) are supported;**
 - 120 threads
 - Each with a PSW (program status word)
- ❑ **PSWs circulate in a control loop**
 - control and data loops pipelined 8 deep
 - PSW in control loop can circulate no faster than data in data loop
 - PSW at queue head fetches and starts execution of next instruction
 - No inter-instruction pipeline forwarding or stalls needed
- ❑ **Clock period: 100 ns**
 - 8 PSWs in control loop => 10 MIPS
 - Maximum perf. per thread => 1.25 MIPS
 - (They tried to sell this as a supercomputer)

(c) 2007 J. E. Smith

49

Processor Organization



(c) 2007 J. E. Smith

50

Processor, contd.

- ❑ **Address space: 32K to 1Mwords (64 bits)**
- ❑ **64 bit instructions**
- ❑ **2048 GP registers + 4096 constants**
 - Registers can be shared among threads
- ❑ **Memory operation**
 - Loads and stores performed by scheduler functional unit (SFU)
 - SFU builds network packet and sends it into switch
 - PSW is removed from control loop and placed in SFU queue
 - PSW is placed back into control loop following memory response
- ❑ **Special operations**
 - control instructions to create/terminate threads
 - full/empty bits in memory and registers
 - busy wait on empty/full operands

(c) 2007 J. E. Smith

51

Switch

- ❑ **Packet switched**
- ❑ **3 bi-directional ports per switch**
 - Every cycle, take in 3 packets, send out 3 packets
- ❑ **"Hot Potato" routing**
 - Form of adaptive routing
 - Do not enqueue on a port conflict
 - Send anyway on another port and raise priority
 - At top priority (15) traverse a circuit through the net

(c) 2007 J. E. Smith

52

Modern Day Multi-Threading

- ❑ **Apply to superscalar pipelines**
 - More resources to share
- ❑ **Also one-wide in-order processors**
 - Provide high efficiency for throughput-oriented servers
- ❑ **Start with Case Study**
 - Intel Pentium 4 *Hyperthreading*

(c) 2007 J. E. Smith

53

Intel Hyperthreading

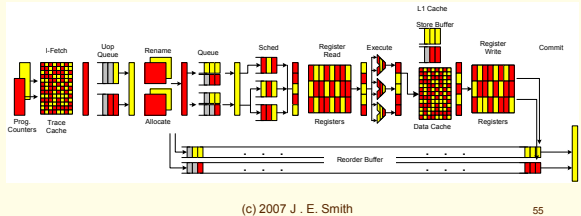
- ❑ **Part of Pentium 4 design (Xeon)**
- ❑ **Two threads per processor**
- ❑ **Goals**
 - Low cost – less than 5% overhead for replicated state
 - Assure forward progress of both threads
 - Make sure both threads get some buffer resources through partitioning or budgeting
 - Single thread running alone does not suffer slowdown

(c) 2007 J. E. Smith

54

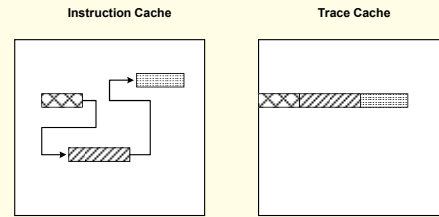
Intel Hyperthreading

- ❑ Main pipeline
 - Pipeline prior to trace cache not shown (see paper)
- ❑ Round-Robin instruction fetching
 - Alternates between threads
 - Avoids dual-ported trace cache
 - BUT trace cache is a shared resource



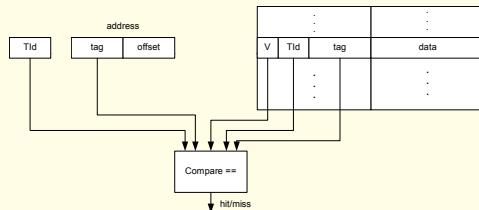
Trace Caches

- ❑ Trace cache captures dynamic traces
- ❑ Increases fetch bandwidth
- ❑ Help shorten pipeline (if predecoded)



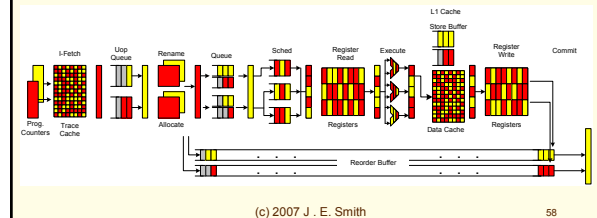
Capacity Resource Sharing

- ❑ Append thread identifier (Tid) to threads in shared capacity (storage) resource
- ❑ Example: cache memory



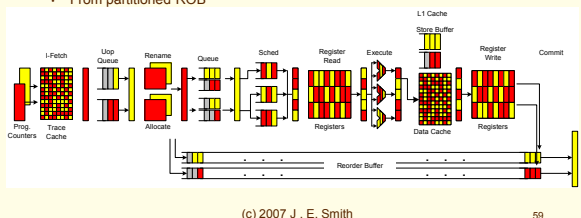
Frontend Implementation

- ❑ Partitioned front-end resources
 - Fetch queue (holds uops)
 - Rename and allocate tables
 - Post-rename queues
- ❑ Partitioning assures forward progress if other thread is blocked
 - Round-robin scheduling



Backend Implementation

- ❑ Physical registers are pooled (shared)
- ❑ Five instruction buffers (schedulers)
 - Shared
 - With an upper limit
- ❑ Instruction issue is irrespective of thread ID
- ❑ Instruction commit is round-robin
 - From partitioned ROB



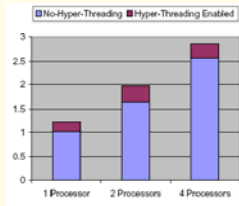
Operating Modes and OS Support

- ❑ MT-mode – two active logical processors; shared/partitioned resources
- ❑ ST-mode (ST0 or ST1) – one logical processor; combined resources
- ❑ HALT – privileged instruction => (normally) low power mode
 - IN MT mode => transition to ST0 or ST1 (depending on the thread that HALTed)
 - In ST mode => low power mode
- ❑ Interrupt to HALTed thread => transition to MT mode
- ❑ OS manages two “processors” (some licensing issues)
 - OS code should HALT rather than idle loop
 - Schedule threads with priority to ST mode (require OS knowledge of hyperthreading)

Performance

OLTP workload

- 21% gain in single and dual systems
- Must be some external bottleneck in 4 processor systems

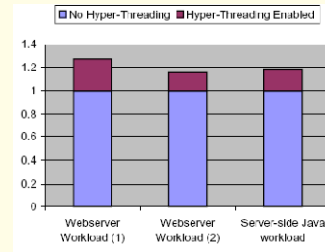


(c) 2007 J. E. Smith

61

Performance

Web server apps

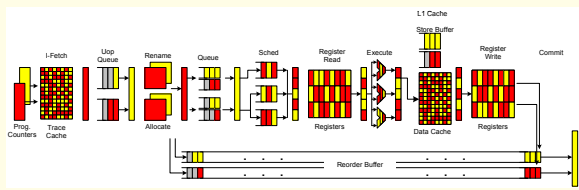


(c) 2007 J. E. Smith

62

Intel Hyperthreading Summary

- Mix of partitioned and shared resources
- Mostly round-robin scheduling
- Primary objective: performance
- Secondary objective: fairness
- Not a lot of obvious structure/reasons for design decisions



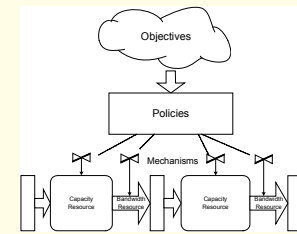
(c) 2007 J. E. Smith

63

Multi-threaded Processors

- Contains both capacity (storage) and bandwidth (processing) resources
- These must be managed in some fashion
 - Can't be done by OS (as with conventional shared hardware resources)
 - Implementation dependent features
 - Fine time granularity

- Objectives
 - Big picture goals
- Policies
 - An algorithmic plan of action
- Mechanisms
 - Implement the policies



(c) 2007 J. E. Smith

64

Objectives

- Optimize aggregate performance
 - Possibly at the expense of individual thread performance
- Provide assured performance for one (or more) threads
 - Soft real-time
 - Fast interactive response time is special-case
- Fairness
 - Often not well-defined (e.g., Intel Pentium4)
 - Often means the lack of obvious unfairness
 - Some quantitative measures have been proposed (next slide)
- Isolation
 - Don't let threads interfere with each other
 - Provides dependable performance
 - Example: shared server

(c) 2007 J. E. Smith

65

Fairness

- Often a secondary goal
- Often defined in terms of performance
 - "Fair" if all threads experience similar slowdowns
 - Compared with running alone
 - $S_i = \text{time}_{\text{unshared}} / \text{time}_{\text{shared}}$
- Harmonic mean speedup
 - $n / \sum(1/S_i)$
 - Captures aggregate performance
 - As well as a measure of fairness
 - A slow thread drags down the harmonic mean more than a fast thread lifts it
- Proportionate slowdown
 - Primary fairness goal
 - Make all the speedups (slowdowns) the same
- Other definitions focus on resources
 - Give each thread a fair share of resources
 - May be equal share (in server example)

(c) 2007 J. E. Smith

66

Scheduling Policies

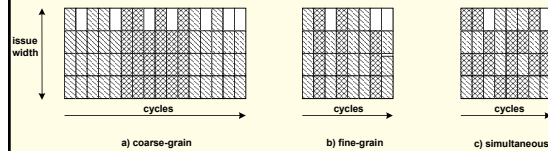
- Applies to allocation of bandwidth resources
 - e.g., pipeline stages
- Granularity
 - When to schedule
- Selection
 - Which to schedule

(c) 2007 J. E. Smith

73

Scheduling Granularity

- Coarse-grain
 - Re-schedule after multiple clock cycles
 - "Switch-on-event"
- Fine-grain
 - Schedule every cycle
 - On thread gets all of resource for that cycle
- Simultaneous
 - Schedule every cycle
 - Resource is shared during that cycle
- Pentium 4 uses combination of fine-grain and simultaneous



(c) 2007 J. E. Smith

74

Thread Selection

- Round-Robin
 - Rotate among threads
- LRU
 - Like round-robin, but if thread has been stalled and re-enters selection process, it goes to front of the line
- FCFS (FIFO)
 - First come – first served – typically applies to a queue
- FR-FCFS
 - First ready FCFS – of those that are ready, the first to arrive gets priority
 - Example: issue buffer
- Priority
 - Top priority thread gets resource
 - May be set by software
 - Maybe established through feedback mechanism in hardware

(c) 2007 J. E. Smith

75

Capacity Policies

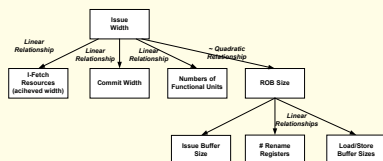
- How big should partitions be?
 - If partitions are flexible
- How much of a shared resource should each thread get?
- Many interesting issues arise with caches
 - To be covered later

(c) 2007 J. E. Smith

76

Sharing Backend Capacity Resources

- Issue Buffer, ROB, Load/Store queues, Physical Registers
 - Grow quadratically with issue width
- Many designs provide resources for high single thread performance
 - To permit single thread mode
 - For good performance when there is only one thread working
 - Implies excess resources when multiple threads

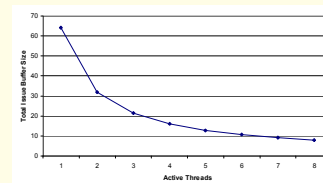


(c) 2007 J. E. Smith

77

Sharing Backend Capacity Resources

- Divide quadratic-growth resource evenly among n threads
 - pipeline width d
- Total resource requirement: $n^*(d/n)^2 = d^2/n$
- More threads => less total resource
- Partitioning may not hurt performance (much)



(c) 2007 J. E. Smith

78

Work Conservation

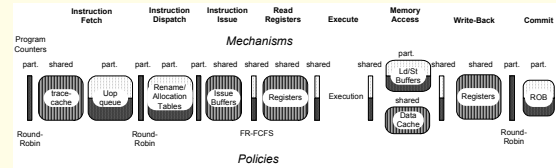
- ❑ If any thread can use a resource, will it be used?
 - If so, then *work conserving*
- ❑ Work conservation is easier with bandwidth resources than capacity (cache) resources
 - With a cache, it is hard to determine if it is being "used"
- ❑ Partitioning leads to non-work conservation
- ❑ Work conservation may be paired with pre-emption

(c) 2007 J. E. Smith

79

Example: Pentium 4

- ❑ Combination of Policies and Mechanisms

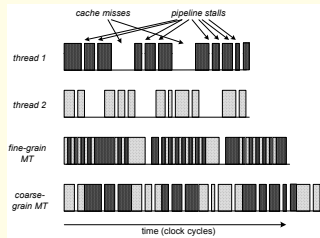


(c) 2007 J. E. Smith

80

Thread Scheduling: In-Order Processors

- ❑ I-Fetch scheduling drives entire pipeline
- ❑ Many small stalls due to register dependences
- ❑ Fine granularity scheduling "meshes" the stalls from multiple threads

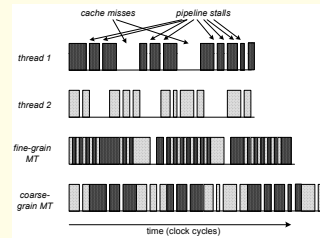


(c) 2007 J. E. Smith

81

Thread Scheduling: In-Order Processors

- ❑ Coarse granularity can also be successful if miss-events are dominant
 - *Switch-on-event* policies; "event" == cache miss

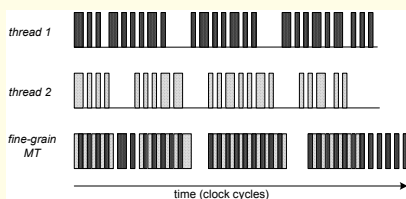


(c) 2007 J. E. Smith

82

Thread Scheduling: In-Order Processors

- ❑ Fine-grain scheduling can reduce or eliminate forwarding paths
 - Single thread performance is diminished
 - Throughput may be unchanged
 - Advantage is in hardware simplicity
 - Example: Denelcor HEP

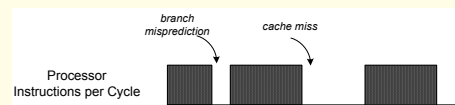


(c) 2007 J. E. Smith

83

Scheduling: Out-of-Order Processors

- ❑ Miss events tend to dominate
- ❑ Out-of-order issue reduces fine-grain stalls
 - In balanced processors
- ❑ *Switch-on-event appears to be good policy*
 - Problems with switch delay
 - For branch misprediction, new thread ready no sooner than old thread
 - Ditto for short (L1) D-cache miss (miss latency ≈ pipeline fill latency)
 - May be useful only for long latency miss events

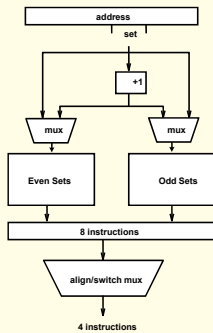


(c) 2007 J. E. Smith

84

O-O-O Scheduling: Fetch Unit

- ❑ **Fetch unit mechanisms**
 - Easier if time partitioned
 - Else dual-ported cache
- ❑ **Easier for narrow fetch widths**
 - Due to discontinuities caused by branches
- ❑ **Wide issue may require dual ported cache, anyway**
 - Or trace cache
- ❑ **Most current processors are narrow enough that partitioning works**
 - Alpha EV-8 (abandoned), 8-wide => two fetches per cycle



(c) 2007 J. E. Smith

85

Fetch Unit Policies

- ❑ **Fine-grain**
- ❑ **Selection establishes policy for entire front-end (up to issue)**
- ❑ **Round-robin with feedback is common (later)**
- ❑ **Also can implement thread priorities**

(c) 2007 J. E. Smith

86

O-O-O Scheduling: Issue Unit

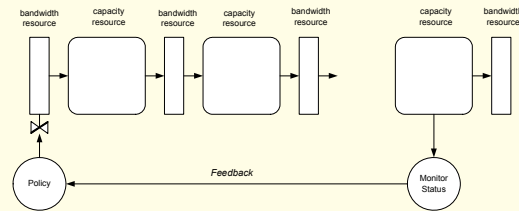
- ❑ **Mechanism allows sharing (no hard partitioning)**
 - Back-end capacity resources are expensive
 - May want one thread to have all of resource
 - May have soft partitioning
- ❑ **Policy is "thread-blind"**
 - Otherwise would be more complicated
 - Typically FR-FCFS
- ❑ **Combination of sharing and thread-blindness could lead to congestion/starvation**
 - A thread with a long cache miss could occupy issue buffer slots and slow other thread(s)
- ❑ **One Solution: Pre-emption on long cache miss**
- ❑ **Another Solution: Feedback to fetch policy**

(c) 2007 J. E. Smith

87

O-O-O Scheduling: Feedback

- ❑ **I-Fetch Policy**
- ❑ **Count in-flight, un-issued instructions for each thread**
 - Give priority to thread with lowest count
 - => ICOUNT policy

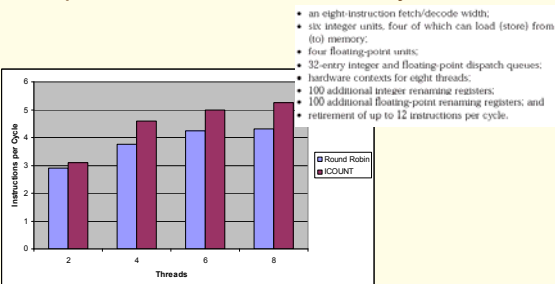


(c) 2007 J. E. Smith

88

ICOUNT

- ❑ **From Tullsen et al.**
- ❑ **Compare ICOUNT with Round-Robin for 8-way SMT**

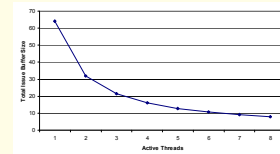


(c) 2007 J. E. Smith

89

Another possibility

- ❑ **From Raasch et al.**
- ❑ **Use soft partition of back-end resources**
 - Soft partition allows single thread mode
- ❑ **Partition doesn't hurt performance much due to quadratic growth**
 - Remember:



- Then use round-robin without feedback (works pretty well)

(c) 2007 J. E. Smith

90

Fairness Policies

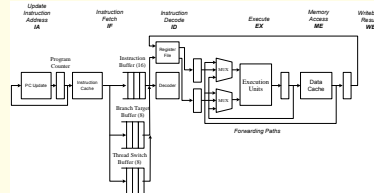
- **Share the pain (proportionate slowdown)**
 - Requires feedback mechanism for estimating “run alone” performance
 - In general, this is not easy
 - Can be done via off-line profiling => coarse granularity policy
- **Share the resources (proportionate resources)**
 - Good for real-time objectives
 - Good for performance isolation
 - Work conservation becomes an issue
 - Leads to “Virtual Private Machines” (later)

(c) 2007 J. E. Smith

91

Case Study: IBM RS64 IV

- **Used in AS/400 series servers (pSeries)**
 - PowerPC ISA
 - Commercial apps, abundant threads
 - => in-order pipeline (for simplicity)
 - High cache miss rates
 - => Switch-on-event (coarse grain) multithreading (for simplicity)



(c) 2007 J. E. Smith

92

Case Study: IBM RS64 IV

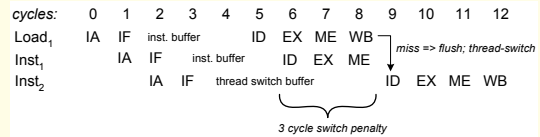
- **Two threads per processor**
 - 4-way superscalar, in-order processor
- **Foreground thread executes until there is a long latency event (e.g., cache miss)**
 - Then switch to the background thread
 - Priorities adjust miss events that cause switches
 - Thread Switch Timeout Register assures fairness when there are few miss event context switches
- **Thread state control register (TSC)**
 - Software-writable, sets priorities
 - Priorities determine events that cause switch
- **Forward Progress Count**
 - Counts number of times a thread was switched in, but made not progress
 - Threshold in TSC
 - If count exceeds threshold, then force forward progress

(c) 2007 J. E. Smith

93

Fast Thread Switches

- **An issue with switch-on-event scheduling**
- **Add thread switch buffer**
 - Use extra l-fetch capacity to fill thread switch buffer with insts. from background thread
 - Also a branch target buffer for quick mispredict recovery
- **Leads to three-cycle switch time**
 - For D-cache, TLB misses
 - Cache miss not detected until WB stage



(c) 2007 J. E. Smith

94

Miss Performance

- **Multithreading can affect miss rates**
 - Long miss (L2) degrades little
 - In server workload there are a lot of thread switches, anyway
 - L1 cache misses degrade little
 - In this workload the miss rate is already high

Table 1 Increase in miss rate because of multithreading.

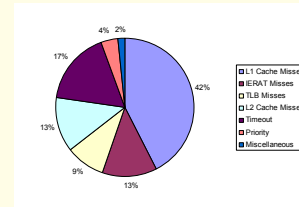
Directory	Increase in miss rate (%)
L2 cache	4.6
L1 I-cache	8.5
L1 D-cache	32.0
TLB	19.0

(c) 2007 J. E. Smith

95

Thread Switch Causes

- **Most thread switches due to L1 misses, then timeouts**



(c) 2007 J. E. Smith

96

Case Study: SUN Niagara

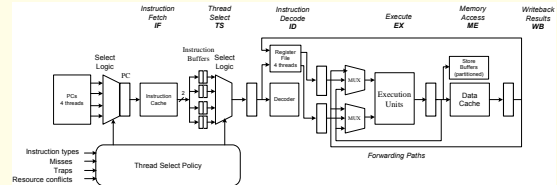
- ❑ **Used in Sun Servers**
 - SPARC ISA
 - Commercial apps, abundant threads
 - => one-wide in-order pipeline (for simplicity, efficiency)
 - Uses fine-grain thread switching
- ❑ **Each chip contains:**
 - Eight cores, four threads each
 - 16K I-cache, 8K D-cache per core
 - Shared floating point unit
 - Shared 3 MByte L2 cache
 - Memory controllers

(c) 2007 J. E. Smith

97

SUN Niagara

- ❑ **In-order Core**
 - Supports four threads
 - Forwarding logic
 - Load has two cycle latency
 - Separate Load/Store buffers per thread
 - Reduces cross-checks, forwarding



(c) 2007 J. E. Smith

98

Pipeline Processing

- ❑ **Coordinated policy at fetch and decode**
- ❑ **Fetch one or two instructions with LRU policy**
 - Dual fetch => unused I-cache slots, for line-fills
- ❑ **Pre-decode identifies long latency instructions**
 - Multiplies, divides, branches, loads
- ❑ **Thread with long latency instruction is temporarily de-selected**
 - For multiplies/divides – until operation is complete
 - For branches – until outcome is known (then no need for prediction)
 - For loads – more complex...

(c) 2007 J. E. Smith

99

Load Instructions

- ❑ **Deselect at least until time data would be available via forward path**
- ❑ **Lower priority until time hit/miss is known**
 - Happens one cycle later than first forward possibility
 - Miss not detected until WB stage
- ❑ **Example:**

cycles:	0	1	2	3	4	5	6	7
load ₀		TS	ID	EX	ME	WB		
add ₁			TS	ID	EX	ME	WB	
load ₁				TS	ID	EX	ME	WB
add ₀					TS	ID	EX	ME WB

hit/miss? forward

(c) 2007 J. E. Smith

100

Case Study: IBM Power5

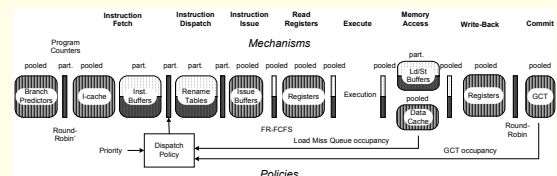
- ❑ **Used in IBM Servers**
 - PowerPC ISA
 - High-end out-of-order superscalar processor
 - Uses simultaneous multi-threading
- ❑ **Each chip contains:**
 - Two cores, two threads each
 - 64K I-cache, 32K D-cache per core
 - Shared 1.875 Mbyte L2 cache
 - Tag directory for external L3 cache
 - Memory controllers
 - System interconnect

(c) 2007 J. E. Smith

101

Power5 Policies and Mechanisms

- ❑ **I-Fetch**
 - Round-robin into partitioned 24-entry fetch buffers
- ❑ **Dispatch Selection**
 - Done via priorities and feedback mechanisms (next slide)
- ❑ **ROB shared via linked list structure**
 - Manages groups of instructions (up to 5) for simpler design



(c) 2007 J. E. Smith

102

Dispatch Policy

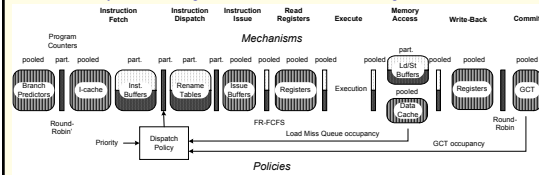
- Primary point for resource management
- Uses software-settable priorities
 - Set via software writeable control register
 - Application software can control some of the priority settings
 - Priority 0 => idle, priority 1 => spinning
 - Both threads at level 1 => throttle to save power
- Hardware feedback can adjust priorities

(c) 2007 J. E. Smith

103

Feedback Priority Adjustment

- Dynamic Resource Balancing
 - Temporarily adjusts priority levels
- GCT (ROB) Occupancy
 - Thread with more than threshold entries gets reduced priority
 - Similar to ICOUNT
- LMQ (load miss queue) Occupancy
 - Thread more than threshold entries gets reduced priority
 - Many outstanding misses => resource blockage

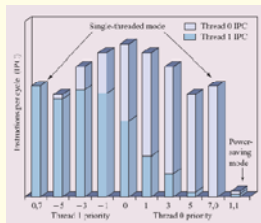


(c) 2007 J. E. Smith

104

Priority Behavior

- Illustration of typical behavior
- Eight priority levels
- Highest aggregate throughput with equal priorities
- Highest single thread performance at priority 7,0



(c) 2007 J. E. Smith

105

SMT "Ugly" Instructions

- Ugly, context-synchronizing instructions
 - System call/return (switch to/from privileged state)
 - Writes to low-level status registers (translation on/off, etc.)
 - TLB invalidates
 - Cache line invalidates
 - Memory barriers (weak consistency)
 - Partial register or flag read/write (not easily renamed)
- Single-threaded CPU:
 - Stall issue, wait for completion, continue
- SMT:
 - This may not be necessary for functional correctness or acceptable for performance
- Need to consider these case by case
- Lots of headaches, potential corner cases

© 2005 Mikko Lipasti

106

Spin Lock Optimization

- Spinning on a lock should not steal resources from other threads
 - Typical synchronization code around critical sections (TTAS):
- ```
spin: ld r0,lock (Ml)(HitS)(MI)
 cmpi c0,r0,0
 bc c0,spin
 ldarx r0,lock (HitS)
 cmpi c0,r0,0
 bc c0,spin
 stdcx r1,lock (DClaim)
 bc c0,spin
...critical section...
 sync
 std r0,lock (DClaim)
```
- Note miss and modified intervention (MI) that precedes falling out of spin loop
  - Could thread switch on failure to acquire lock (test-and-test-and-set is not very amenable to that)

© 2005 Mikko Lipasti

107

## Alpha 21464 Sleep Instruction

- Alpha 21464 proposed clever new sleep instruction that puts thread to sleep until address changes
  - Here's the new spin lock code:
- ```
ld    r0,lock    # test lock
cmpi  c0,r0,0
bc   c0,test    # branch to load-locked
spin: sleep    lock    # wait until lock changes
      ld    r0,lock    # (i.e. snoop invalidate)
      cmpi  c0,r0,0
      bc   c0,spin
test: ldarx   r0,lock    # do atomic test-and-set
      cmpi  c0,r0,0
      bc   c0,spin    # go to spin loop
      stdcx r1,lock    # go to spin loop
      bc   c0,spin
...critical section...
      sync
      std   r0,lock    # release lock
```

© 2005 Mikko Lipasti

108

SMT Consistency Model Issues

- **Consistency model restricts when other processors can see your write**
 - What does "other processor" mean in an MT processor?
 - Descriptions of consistency models may need to be updated...
 - Probably "other thread" is sufficient
- **Weak consistency (e.g. PowerPC)**
 - Stores visible anytime, so store forwarding or reading stored lines from cache early is OK
 - Memory barriers (sync) cause problems: store queue flush
- **Sequential consistency**
 - Stores must be atomically visible to all other processors (write atomicity)
 - Cannot forward store values until preceding stores are done
 - Need logically separate store queues (or thread IDs in store queue entries)
 - Cannot retire stores to L1 until coherence activity is done
- **Other issues? Certainly there is no exhaustive treatment in the literature.**