Out-of-order Superscalar

740
October 31, 2014

So Far …

CPU Time = $CPU\ clock\ cycles \times clock\ cycle\ time$

$= \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{seconds}{cycle}$

- Increase clock frequency
- Decrease CPI
  - Pipeline:
    - divide CPI by # of stages + bubbles
    - add bypass
  - Superscalar
    - divide CPI by issue width + bubbles from
      - » data dependencies
      - » control dependencies

What next?

- What slows us down here?

```
// for (i=0; i<N; i++) c[i] = a[i]*b[i]+i*2;

loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```


What next?

- What slows us down here?

```
// for (i=0; i<N; i++) c[i] = a[i]*b[i]+i*2;

loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

How can we improve this?

- *Assume:*
  - 2-issue superscalar
  - 1 LD/ST unit (2 cycles),
  - 1 mult (2 cycles)
  - 1 add (1 cycle)

```
loop: ld     r1, (r2+r9*8)
      ld     r3, (r4+r9*8)
      mult   r5, r3, r1
      mult   r3, r9, 2
      add    r5, r5, r3
      st     r5, (r6+r9*8)
      add    r9, r9, 1
      cmp    r9, r10
      bnz    loop
```

```
ld       r1, (r2+r9*8)
..ld       r3, (r4+r9*8)

  ..mult    r5, r3, r1
    ..mult    r3, r9, 2

      ..add     r5, r5, r3
        .st       r5, (r6+r9*8)

         .add     r9, r9, 1
          .cmp     r9, r10

           .bnz      loop
```

12 cycles

What next?

- What slows us down here?

```
loop: ld     r1, (r2+r9*8)
      ld     r3, (r4+r9*8)
      mult   r5, r3, r1
      mult   r3, r9, 2
      add    r5, r5, r3
      st     r5, (r6+r9*8)
      add    r9, r9, 1
      cmp    r9, r10
      bnz    loop
```

- Data Dependencies
  - RAW: Read after write. (true dependence)

What next?

- What slows us down here?

```
loop: ld     r1, (r2+r9*8)
      ld     r3, (r4+r9*8)
      mult   r5, r3, r1
      mult   r3, r9, 2
      add    r5, r5, r3
      st     r5, (r6+r9*8)
      add    r9, r9, 1
      cmp    r9, r10
      bnz    loop
```

- Data Dependencies
  - RAW: Read after write. (true dependence)
  - WAR: write after read (false dependence)

What next?

- What slows us down here?

```
loop: ld     r1, (r2+r9*8)
      ld     r3, (r4+r9*8)
      mult   r5, r3, r1
      mult   r3, r9, 2
      add    r5, r5, r3
      st     r5, (r6+r9*8)
      add    r9, r9, 1
      cmp    r9, r10
      bnz    loop
```

- Data Dependencies
  - RAW: Read after write. (true dependence)
  - WAR: write after read (false dependence)
  - WAW: write after write (false dependence)

Eliminating WAR (and WAW)

- What slows us down here?

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r11, r9, 2
      add   r5, r5, r11
      st    r5, (r6+r9*8)
      add   r12, r9, 1
      cmp   r12, r10
      mov   r9, r12
      bnz   loop
```

- Data Dependencies
  - RAW: Read after write. (true dependence)
  - WAR: write after read (false dependence)
  - WAW: write after write (false dependence)

Reschedule

```
loop: ld    r1, (r2+r9*8)
      mult  r11, r9, 2
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r11, r9, 2   (struck out)
      add   r5, r5, r11
      st    r5, (r6+r9*8)
      add   r12, r9, 1
      cmp   r12, r10
      mov   r9, r12
      bnz   loop
```


Can we do better?

```
ld      r1, (r2+r9*8)
mult    r11, r9, 2

  ld      r3, (r4+r9*8)
    mult    r5, r3, r1

      add     r5, r5, r11
       st      r5, (r6+r9*8)

       add     r12, r9, 1
        cmp     r12, r10

         mov     r9, r12
         bnz     loop
```

```
loop: ld    r1, (r2+r9*8)
      mult  r11, r9, 2
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      add   r5, r5, r11
      st    r5, (r6+r9*8)
      add   r12, r9, 1
      cmp   r12, r10
      mov   r9, r12
      bnz   loop
```

10 cycles

More Rescheduling

```
ld      r1, (r2+r9*8)
mult    r11, r9, 2

  ld      r3, (r4+r9*8)
  add     r12, r9, 1

   cmp     r12, r10
   mult    r5, r3, r1

     add     r5, r5, r11
      st      r5, (r6+r9*8)

       mov     r9, r12
       bnz     loop
```

```
loop: ld    r1, (r2+r9*8)
      mult  r11, r9, 2
      ld    r3, (r4+r9*8)
      add   r12, r9, 1
      cmp   r12, r10
      mult  r5, r3, r1
      add   r5, r5, r11
      st    r5, (r6+r9*8)
      mov   r9, r12
      bnz   loop
```

7 cycles

What Is Holding Us Back?

```
ld       r1, (r2+r9*8)
mult     r11, r9, 2

  ld       r3, (r4+r9*8)
  add      r12, r9, 1

   cmp      r12, r10
   mult     r5, r3, r1

     add      r5, r5, r11
      st       r5, (r6+r9*8)

       mov      r9, r12
       bnz      loop
```

- In order issue
- Not enough Function units
- Function units too slow
- Not enough register names
- Control dependence
- Static scheduling

What Is Holding Us Back?

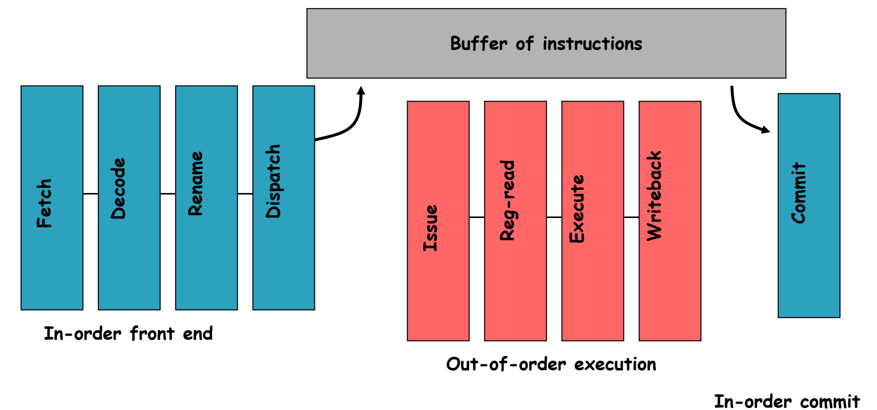
```
ld       r1, (r2+r9*8)
mult     r11, r9, 2

  ld       r3, (r4+r9*8)
  add      r12, r9, 1

   cmp      r12, r10
   mult     r5, r3, r1

     add      r5, r5, r11
      st       r5, (r6+r9*8)

       bnz      loop
       ld       r1, (r2+r12*8)
```

- In order issue
- Not enough Function units
- Function units too slow
- Not enough register names
- Control dependence
- static scheduling

Out-of-Order Superscalar

- In-order fetch & decode
- Out-of-order
  - issue
  - register-read
  - execute
  - memory
  - write-back
- In-order commit

Out-of-Order Pipeline



Out-of-Order Execution

- Also called "Dynamic scheduling"
  - Done by the hardware on-the-fly during execution
- Looks at a "window" of instructions waiting to execute
  - Each cycle, picks the next ready instruction(s)
- Two steps to enable out-of-order execution:
  - Step #1: Register renaming – to avoid "false" dependencies
  - Step #2: Dynamically schedule – to enforce "true" dependencies
- Key to understanding out-of-order execution:
  - **Data dependencies**

Dependence types

- **RAW** (Read After Write) = "true dependence" (true)
  - mul r0 * r1 ➔ r2
  - …
  - add r2 + r3 ➔ r4
- **WAW** (Write After Write) = "output dependence" (false)
  - mul r0 * r1➔ r2
  - …
  - add r1 + r3 ➔ r2
- **WAR** (Write After Read) = "anti-dependence" (false)
  - mul r0 * r1 ➔ r2
  - …
  - add r3 + r4 ➔ r1
- WAW & WAR are "false", Can be **totally eliminated** by "renaming"

Step #1: Register Renaming

- To eliminate register conflicts/hazards
- "Architected" vs "Physical" registers – level of indirection
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1**→**p1**, **r2**→**p2**, **r3**→**p3**, **p4–p7** are "available"

MapTable r1	r2	r3	FreeList	Original insns	Renamed insns
p1	p2	p3	p4,p5,p6,p7	add r2,r3➔r1	add p2,p3➔p4
p4	p2	p3	p5,p6,p7	sub r2,r1➔r3	sub p2,p4➔p5
p4	p2	p5	p6,p7	mul r2,r3➔r3	mul p2,p5➔p6
p4	p2	p6	p7	div r1,4➔r1	div p4,4➔p7

- Renaming – conceptually write each register once
  - \+ Removes **false** dependences
  - \+ Leaves **true** dependences intact!
- When to reuse a physical register? After overwriting insn done

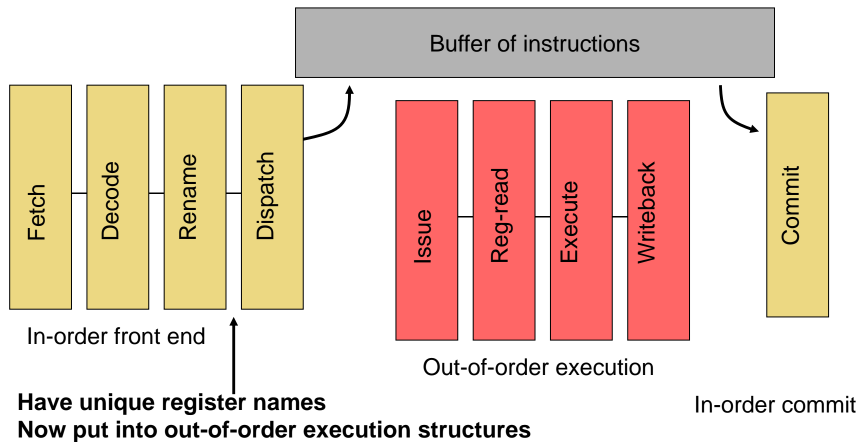
Register Renaming Algorithm

- Two key data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)
- Algorithm: at "decode" stage for each instruction:

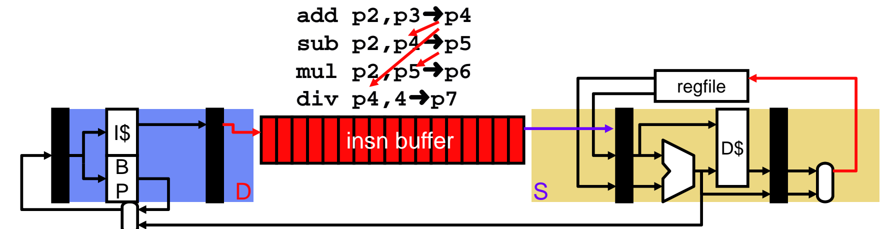
```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.old_phys_output = maptable[insn.arch_output]
new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

- At "commit"
  - Once all older instructions have committed, free register `free_phys_reg(insn.old_phys_output)`

Out-of-order Pipeline



Step #2: Dynamic Scheduling



Ready Table (Time ↓)

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

```
add p2,p3➔p4
sub p2,p4➔p5 and div p4,4➔p7
mul p2,p5➔p6
```

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute oldest "ready" instruction, set output as "ready"

Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] ➔ yes/no (part of "issue queue")
- Algorithm at "issue" stage (prior to read registers):

```
foreach instruction:
   if table[insn.phys_input1] == ready &&
       table[insn.phys_input2] == ready then
            insn is "ready"
select the oldest "ready" instruction
   table[insn.phys_output] = ready
```

- Multiple-cycle instructions? (such as loads)
  - For an insn with latency of N, set "ready" bit N-1 cycles in future

Cycle 0

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop

ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
```

Arch	Phys
r1	p1
r2	p2
r3	p3
r4	p4
r5	
r6	p6
r7	
r8	
r9	p9
r10	p10

Reg	value	ready?
p1		n
p2	a	y
p3		n
p4	b	y
p5		
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11		
p12		
p13		
p14		

Cycle 1

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

```
ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
    mult    p5, p3, p1
      mult    p11, p9, 2
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p5
r6	p6
r7	
r8	
r9	p9
r10	p10

Reg	value	ready?
p1		n
p2	a	y
p3		n
p4	b	y
p5	*	n
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11	*	n
p12		
p13		
p14		

Cycle 2

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

```
ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
    mult    p5, p3, p1
      mult    p11, p9, 2
        add     p12, p5, p11
          st      p12, (p6+p9*8)
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p12
r6	p6
r7	
r8	
r9	p9
r10	p10

Reg	value	ready?
p1		y
p2	a	y
p3		n
p4	b	y
p5	*	n
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11	*	n
p12		
p13		
p14		

Cycle 3

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

```
ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
    mult    p5, p3, p1
      mult    p11, p9, 2
        add     p12, p5, p11
          st      p12, (p6+p9*8)
   add     p13, p9, 1
    cmp     p13, p10
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p12
r6	p6
r7	
r8	
r9	p13
r10	p10

Reg	value	ready?
p1		y
p2	a	y
p3		n
p4	b	y
p5	*	n
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11	*	n
p12		
p13	+	n
p14		

Cycle 4

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

```
ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
    mult    p5, p3, p1
      mult    p11, p9, 2
        add     p12, p5, p11
          st      p12, (p6+p9*8)
   add     p13, p9, 1
    cmp     p13, p10
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p12
r6	p6
r7	
r8	
r9	p13
r10	p10

Reg	value	ready?
p1		y
p2	a	y
p3		y
p4	b	y
p5	*	n
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11	*	n
p12		
p13	+	y
p14		

Cycle 5

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p12
r6	p6
r7	
r8	
r9	p13
r10	p10

Reg	value	ready?
p1		y
p2	a	y
p3		y
p4	b	y
p5	*	n
p6	c	y
p7		
p8		
p9	i	y
p10	n	y
p11	*	n
p12		
p13	+	y
p14		

```
ld    p1, (p2+p9*8)
  ld    p3, (p4+p9*8)
    mult    p5, p3, p1
      mult    p11, p9, 2
        add     p12, p5, p11
          st      p12, (p6+p9*8)
   add     p13, p9, 1
    cmp     p13, p10
     bnz     loop
```


Cycle 6

```
loop: ld    r1, (r2+r9*8)
      ld    r3, (r4+r9*8)
      mult  r5, r3, r1
      mult  r3, r9, 2
      add   r5, r5, r3
      st    r5, (r6+r9*8)
      add   r9, r9, 1
      cmp   r9, r10
      bnz   loop
```

Arch	Phys
r1	p1
r2	p2
r3	p11
r4	p4
r5	p12
r6	p6
r7	p7
r8	p8
r9	p13
r10	p10

Reg	value	ready?
p1		y
p2	a	y
p3		y
p4	b	y
p5	*	n
p6	c	y
p7		n
p8		n
p9	i	y
p10	n	y
p11	*	n
p12		
p13	+	y
p14		

```
mult    p5, p3, p1
      mult    p11, p9, 2
        add     p12, p5, p11
         st      p12, (p6+p9*8)
   add     p13, p9, 1
    cmp     p13, p10
     bnz     loop
      ld      p7, (p2+p13*8)
      ld      p8, (p4+p13*8)
```


Register Renaming

Register Renaming Algorithm (Simplified)

- Two key data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)
- Algorithm: at "decode" stage for each instruction:

```
insn.phys_input1 =
  maptable[insn.arch_input1]
insn.phys_input2 =
  maptable[insn.arch_input2]

new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

Renaming example

xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

Renaming example

xor **r1** ^ **r2** ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

→ xor **p1** ^ **p2** ➜

Map table

r1	**p1**
r2	**p2**
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

Renaming example

xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

→ xor p1 ^ p2 ➜ **p6**

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

Renaming example

xor r1 ^ r2 ➜ **r3**
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1

→ xor p1 ^ p2 ➜ p6

Map table

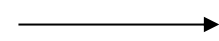
r1	p1
r2	p2
r3	**p6**
r4	p4
r5	p5

Free-list

p7
p8
p9
p10

Renaming example

xor r1 ^ r2 ➔ r3
add **r3** + **r4** ➔ r4
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add **p6** + **p4** ➔

r1	p1
r2	p2
r3	**p6**
r4	**p4**
r5	p5

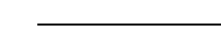
Map table

p7
p8
p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ **p7**

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

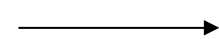
Map table

p7
p8
p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ **r4**
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7

r1	p1
r2	p2
r3	p6
r4	**p7**
r5	p5

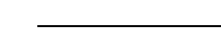
Map table

p8
p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub **r5** - **r2** ➔ r3
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub **p5** - **p2** ➔

r1	p1
r2	**p2**
r3	p6
r4	p7
r5	**p5**

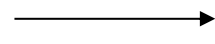
Map table

p8
p9
p10

Free-list

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ **p8**

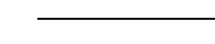
r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

Free-list: **p8**, p9, p10

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ **r3**
addi r3 + 1 ➔ r1



xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8

r1	p1
r2	p2
r3	**p8**
r4	p7
r5	p5

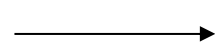
Map table

Free-list: p9, p10

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ r3
addi **r3** + 1 ➔ r1

xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8
addi **p8** + 1 ➔



r1	p1
r2	p2
r3	**p8**
r4	p7
r5	p5

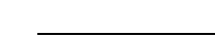
Map table

Free-list: p9, p10

Renaming example

xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1

xor p1 ^ p2 ➔ p6
add p6 + p4 ➔ p7
sub p5 - p2 ➔ p8
addi p8 + 1 ➔ **p9**



r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: **p9**, p10

Renaming example

xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ **r1**

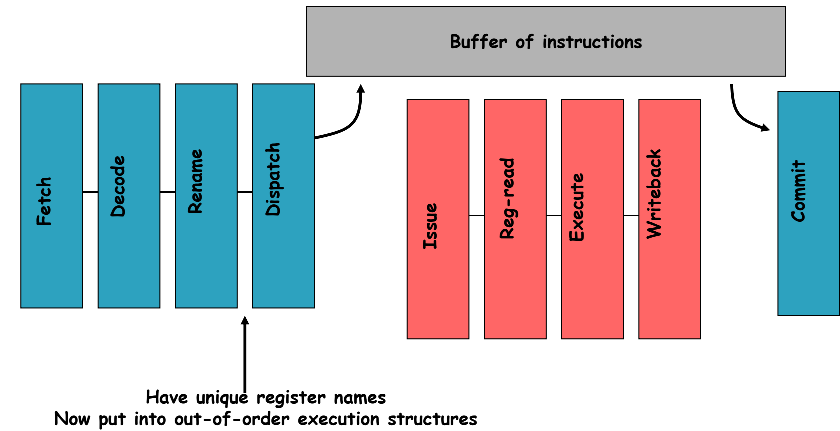
xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9

r1	**p9**
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list

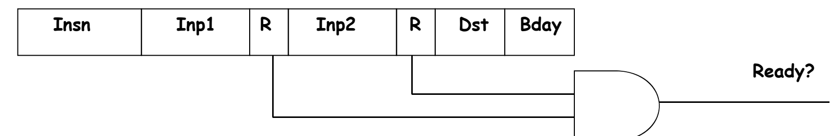
Out-of-order Pipeline



Dynamic Scheduling Mechanisms

Dispatch

- Put renamed instructions into out-of-order structures
- Re-order buffer (ROB)
  - Holds instructions until commit
- Issue Queue
  - Central piece of scheduling logic
  - Holds un-executed instructions
  - Tracks ready inputs
    - Physical register names + ready bit
    - "AND" the bits to tell if ready



Dispatch Steps

- Allocate Issue Queue (IQ) slot
  - Full? Stall
- Read **ready bits** of inputs
  - 1-bit per physical reg
- Clear **ready bit** of output in table
  - Instruction has not produced value yet
- Write data into Issue Queue (IQ) slot

Dispatch Example

xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Bday

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	y
p7	y
p8	y
p9	y

Dispatch Example

xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Bday
xor	p1	y	p2	y	p6	0

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	y
p8	y
p9	y

Dispatch Example

xor p1 ^ p2 ➜ p6
add p6 + p4 ➜ p7
sub p5 - p2 ➜ p8
addi p8 + 1 ➜ p9

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Bday
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	y
p9	y

Dispatch Example

```
xor  p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9
```

Issue Queue

Insn	Inp1	R	Inp2	R	Dst	Bday
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	y

Dispatch Example

```
xor  p1 ^ p2 → p6
add p6 + p4 → p7
sub p5 - p2 → p8
addi p8 + 1 → p9
```

Issue Queue

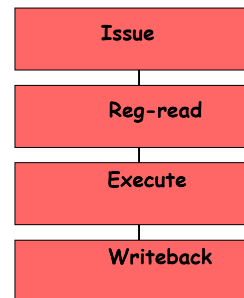
Insn	Inp1	R	Inp2	R	Dst	Bday
xor	p1	y	p2	y	p6	0
add	p6	n	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	n	---	y	p9	3

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	n
p8	n
p9	n

Out-of-order pipeline

- Execution (out-of-order) stages
- **Select** ready instructions
  - Send for execution
- **Wakeup** dependents



Dynamic Scheduling/Issue Algorithm

- Data structures:
  - Ready table[phys_reg] → yes/no (part of issue queue)
- Algorithm at "schedule" stage (prior to read registers):

```
foreach instruction:
  if table[insn.phys_input1] == ready &&
     table[insn.phys_input2] == ready then
       insn is "ready"
select the oldest "ready" instruction
  table[insn.phys_output] = ready
```

Issue = Select + Wakeup

- **Select** oldest of "ready" instructions
  - ➢"xor" is the oldest ready instruction below
  - ➢"xor" and "sub" are the two oldest ready instructions below
  - Note: may have resource constraints: i.e. load/store/floating point

Insn	Inp1	R	Inp2	R	Dst	Bday	
xor	p1	y	p2	y	p6	0	Ready!
add	p6	n	p4	y	p7	1	
sub	p5	y	p2	y	p8	2	Ready!
addi	p8	n	---	y	p9	3	

Issue = Select + Wakeup

- Wakeup dependent instructions
  - Search for destination (Dst) in inputs & set "ready" bit
    - Implemented with a special memory array circuit called a Content Addressable Memory (CAM)
  - Also update ready-bit table for future instructions

Insn	Inp1	R	Inp2	R	Dst	Bday
xor	p1	y	p2	y	p6	0
add	p6	y	p4	y	p7	1
sub	p5	y	p2	y	p8	2
addi	p8	y	---	y	p9	3

Ready bits

p1	y
p2	y
p3	y
p4	y
p5	y
p6	y
p7	n
p8	y
p9	n

- For multi-cycle operations (loads, floating point)
  - Wakeup deferred a few cycles
  - Include checks to avoid structural hazards

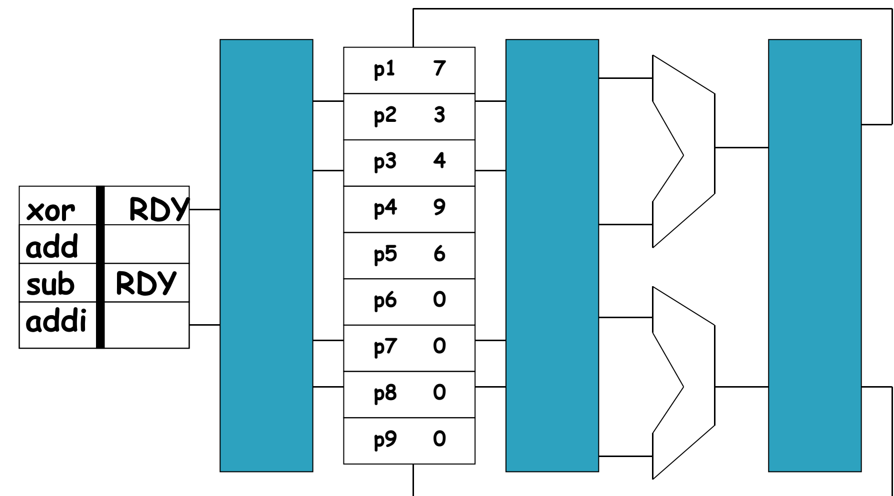
Issue

- **Select/Wakeup** one cycle
- Dependent instructions execute on back-to-back cycles
  - Next cycle: add/addi are ready:

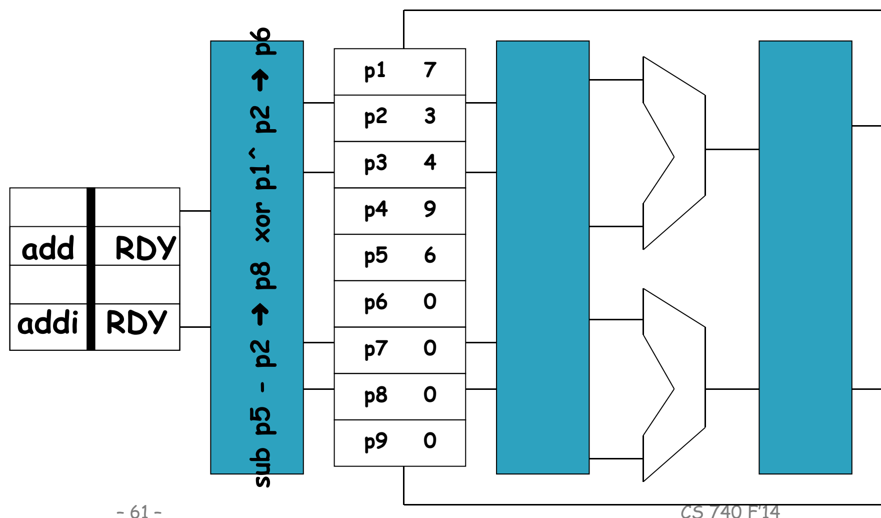
Insn	Inp1	R	Inp2	R	Dst	Bday
add	p6	y	p4	y	p7	1
addi	p8	y	---	y	p9	3

- Issued instructions are removed from issue queue
  - Free up space for subsequent instructions

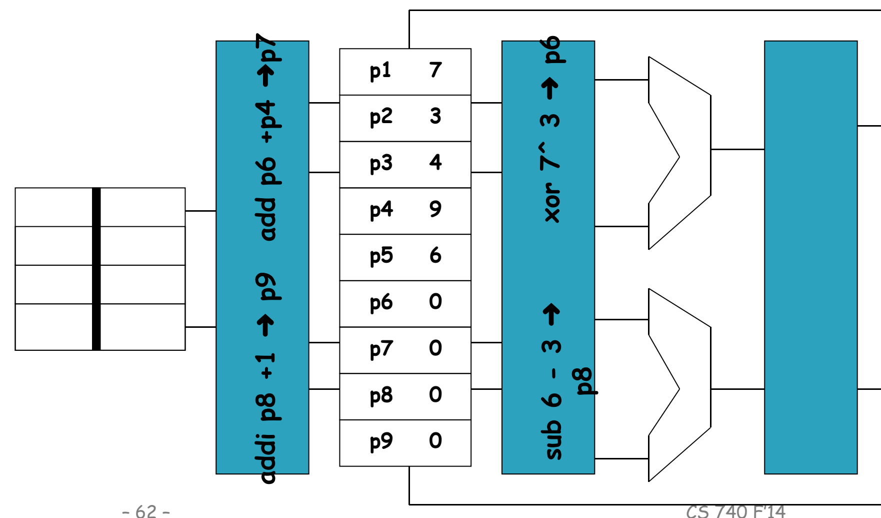
OOO execution (2-wide)



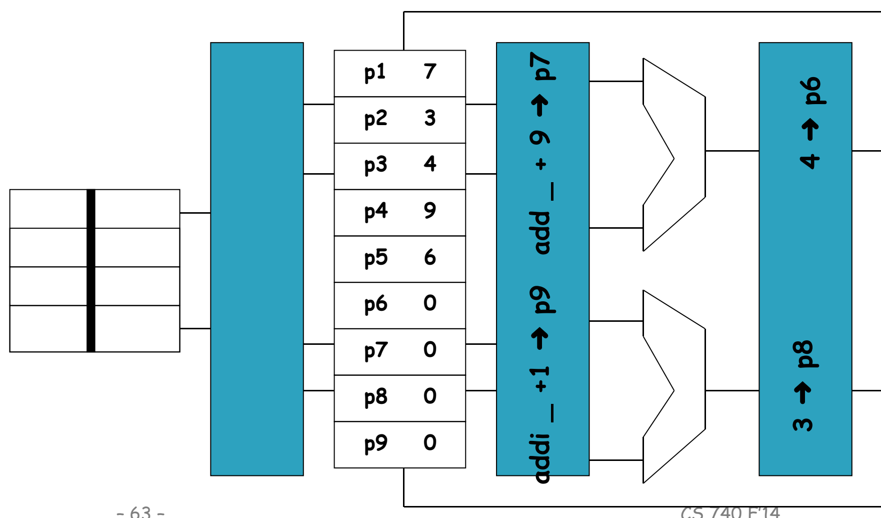
OOO execution (2-wide)



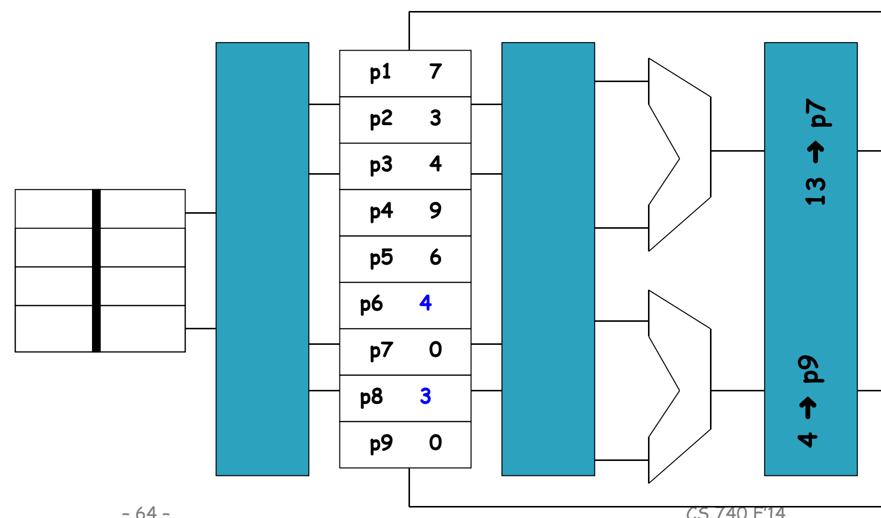
OOO execution (2-wide)



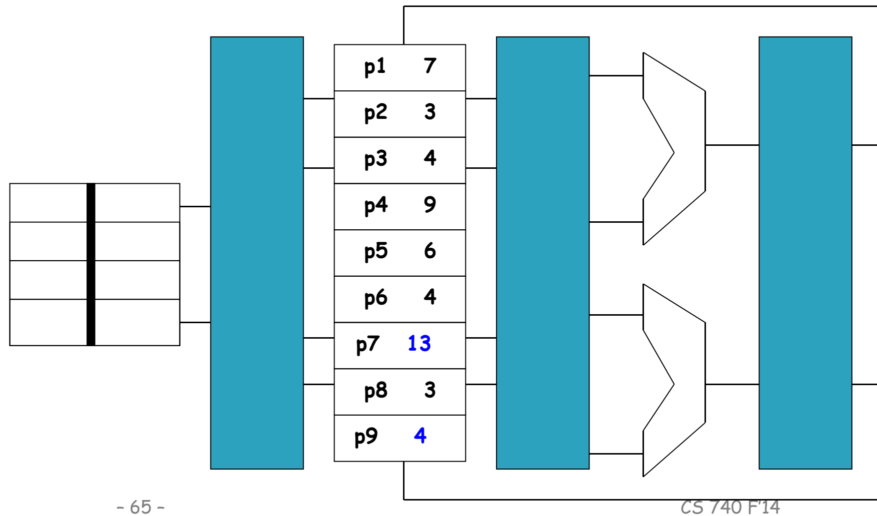
OOO execution (2-wide)



OOO execution (2-wide)



OOO execution (2-wide)



When Does Register Read Occur?

- Current approach: after select, right before execute
  - **Not during in-order part of pipeline, in out-of-order part**
  - Read **physical** register (renamed)
  - Or get value via bypassing (based on physical register name)
  - This is Pentium 4, MIPS R10k, Alpha 21264, IBM Power4, Intel's "Sandy Bridge" (2011)
  - Physical register file may be large
    - Multi-cycle read
- Older approach:
  - Read as part of "issue" stage, keep values in Issue Queue
    - At commit, write them back to "architectural register file"
  - Pentium Pro, Core 2, Core i7
  - Simpler, but may be less energy efficient (more data movement)

Renaming Revisited

Re-order Buffer (ROB)

- ROB entry holds all info for recovery/commit
  - **All instructions** & in order
  - Architectural register names, physical register names, insn type
  - Not removed until very last thing ("commit")
- Operation
  - Dispatch: insert at tail (if full, stall)
  - Commit: remove from head (if not yet done, stall)
- Purpose: tracking for in-order commit
  - » Maintain appearance of in-order execution
  - » Done to support:
  - **Misprediction recovery**
  - **Freeing of physical registers**

Renaming revisited

- Track (or "log") the "overwritten register" in ROB
  - Free this register at commit
  - Also used to restore the map table on "recovery"
    - Branch mis-prediction recovery

Register Renaming Algorithm (Full)

- Two key data structures:
  - maptable[architectural_reg] ➔ physical_reg
  - Free list: allocate (new) & free registers (implemented as a queue)
- Algorithm: at "decode" stage for each instruction:

```
insn.phys_input1 = maptable[insn.arch_input1]
insn.phys_input2 = maptable[insn.arch_input2]
insn.old_phys_output = maptable[insn.arch_output]
new_reg = new_phys_reg()
maptable[insn.arch_output] = new_reg
insn.phys_output = new_reg
```

- **At "commit"**
  - **Once all older instructions have committed, free register**
    `free_phys_reg(insn. old_phys_output)`

Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers
- How to restore map table?
  - Option #1: log-based reverse renaming to recover each instruction
    - Tracks the old mapping to allow it to be reversed
    - Done sequentially for each instruction (slow)
    - See next slides
  - Option #2: checkpoint-based recovery
    - Checkpoint state of maptable and free list each cycle
    - Faster recovery, but requires more state
  - Option #3: hybrid (checkpoint for branches, unwind for others)

Renaming example

```
xor r1 ^ r2 ➔ r3
add r3 + r4 ➔ r4
sub r5 - r2 ➔ r3
addi r3 + 1 ➔ r1
```

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

Renaming example

```
xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1
```

→ xor p1 ^ p2 ➜ [p3]

Map table

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list

p6
p7
p8
p9
p10

Renaming example

```
xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1
```

→ xor p1 ^ p2 ➜ p6 [p3]

Map table

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Free-list

p7
p8
p9
p10

Renaming example

```
xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1
```

xor p1 ^ p2 ➜ p6 [p3]
→ add p6 + p4 ➜ [p4]

Map table

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Free-list

p7
p8
p9
p10

Renaming example

```
xor r1 ^ r2 ➜ r3
add r3 + r4 ➜ r4
sub r5 - r2 ➜ r3
addi r3 + 1 ➜ r1
```

xor p1 ^ p2 ➜ p6 [p3]
→ add p6 + p4 ➜ p7 [p4]

Map table

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Free-list

p8
p9
p10

Renaming example

```
xor r1 ^ r2 ➔ r3          xor  p1 ^ p2 ➔ p6        [ p3 ]
add r3 + r4 ➔ r4          add p6 + p4 ➔ p7         [ p4 ]
sub r5 - r2 ➔ r3           sub p5 - p2 ➔           [ p6 ]
addi r3 + 1 ➔ r1
```

Map table

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Free-list

p8
p9
p10

Renaming example

```
xor r1 ^ r2 ➔ r3          xor  p1 ^ p2 ➔ p6        [ p3 ]
add r3 + r4 ➔ r4          add p6 + p4 ➔ p7         [ p4 ]
sub r5 - r2 ➔ r3          sub p5 - p2 ➔ p8         [ p6 ]
addi r3 + 1 ➔ r1
```

Map table

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p9
p10

Renaming example

```
xor r1 ^ r2 ➔ r3          xor  p1 ^ p2 ➔ p6        [ p3 ]
add r3 + r4 ➔ r4          add p6 + p4 ➔ p7         [ p4 ]
sub r5 - r2 ➔ r3          sub p5 - p2 ➔ p8         [ p6 ]
addi r3 + 1 ➔ r1            addi p8 + 1 ➔          [ p1 ]
```

Map table

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p9
p10

Renaming example

```
xor r1 ^ r2 ➔ r3          xor  p1 ^ p2 ➔ p6        [ p3 ]
add r3 + r4 ➔ r4          add p6 + p4 ➔ p7         [ p4 ]
sub r5 - r2 ➔ r3          sub p5 - p2 ➔ p8         [ p6 ]
addi r3 + 1 ➔ r1          addi p8 + 1 ➔ p9         [ p1 ]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

Recovery Example

Now, let's use this info. to recover from a branch misprediction

```
bnz r1 loop              bnz p1, loop          [    ]
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6      [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7      [ p4 ]
sub r5 - r2 → r3         sub p5 - p2 → p8      [ p6 ]
addi r3 + 1 → r1         addi p8 + 1 → p9      [ p1 ]
```

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p10

Recovery Example

```
bnz r1 loop              bnz p1, loop          [    ]
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6      [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7      [ p4 ]
sub r5 - r2 → r3         sub p5 - p2 → p8      [ p6 ]
addi r3 + 1 → r1         addi p8 + 1 → p9      [ p1 ]
```

r1	p1
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p9, p10

Recovery Example

```
bnz r1 loop              bnz p1, loop          [    ]
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6      [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7      [ p4 ]
sub r5 - r2 → r3         sub p5 - p2 → p8      [ p6 ]
```

r1	p1
r2	p2
r3	p6
r4	p7
r5	p5

Map table

Free-list: p8, p9, p10

Recovery Example

```
bnz r1 loop              bnz p1, loop          [    ]
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6      [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7      [ p4 ]
```

r1	p1
r2	p2
r3	p6
r4	p4
r5	p5

Map table

Free-list: p7, p8, p9, p10

Recovery Example

```
bnz r1 loop              bnz p1, loop            [    ]
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6        [ p3 ]
```

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

Free-list
p6
p7
p8
p9
p10

Free-list

Recovery Example

```
bnz r1 loop              bnz p1, loop            [    ]
```

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

Free-list
p6
p7
p8
p9
p10

Free-list

Commit

```
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6        [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7        [ p4 ]
sub r5 - r2 → r3         sub p5 - p2 → p8        [ p6 ]
addi r3 + 1 → r1         addi p8 + 1 → p9        [ p1 ]
```

- **Commit: instruction becomes architected state**
  - **In-order, only when instructions are finished**
  - **Free overwritten register (why?)**

Freeing over-written register

```
xor r1 ^ r2 → r3         xor p1 ^ p2 → p6        [ p3 ]
add r3 + r4 → r4         add p6 + p4 → p7        [ p4 ]
sub r5 - r2 → r3         sub p5 - p2 → p8        [ p6 ]
addi r3 + 1 → r1         addi p8 + 1 → p9        [ p1 ]
```

- P3 was r3 before xor
- P6 is r3 after xor
  - Anything older than xor should read p3
  - Anything younger than xor should read p6 (until another insn writes r3)
- At commit of xor, no older instructions exist

Commit Example

```
xor r1 ^ r2 ➔ r3        xor  p1 ^ p2 ➔ p6      [ p3 ]
add r3 + r4 ➔ r4        add p6 + p4 ➔ p7       [ p4 ]
sub r5 - r2 ➔ r3        sub p5 - p2 ➔ p8       [ p6 ]
addi r3 + 1 ➔ r1        addi p8 + 1 ➔ p9       [ p1 ]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10

Commit Example

```
xor r1 ^ r2 ➔ r3        xor  p1 ^ p2 ➔ p6      [ p3 ]
add r3 + r4 ➔ r4        add p6 + p4 ➔ p7       [ p4 ]
sub r5 - r2 ➔ r3        sub p5 - p2 ➔ p8       [ p6 ]
addi r3 + 1 ➔ r1        addi p8 + 1 ➔ p9       [ p1 ]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3

Commit Example

```
add r3 + r4 ➔ r4        add p6 + p4 ➔ p7       [ p4 ]
sub r5 - r2 ➔ r3        sub p5 - p2 ➔ p8       [ p6 ]
addi r3 + 1 ➔ r1        addi p8 + 1 ➔ p9       [ p1 ]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3
p4

Commit Example

```
sub r5 - r2 ➔ r3        sub p5 - p2 ➔ p8       [ p6 ]
addi r3 + 1 ➔ r1        addi p8 + 1 ➔ p9       [ p1 ]
```

Map table

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Free-list

p10
p3
p4
p6

Commit Example

addi r3 + 1 ➜ r1 addi p8 + 1 ➜ p9 [p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6
p1

Free-list

Commit Example

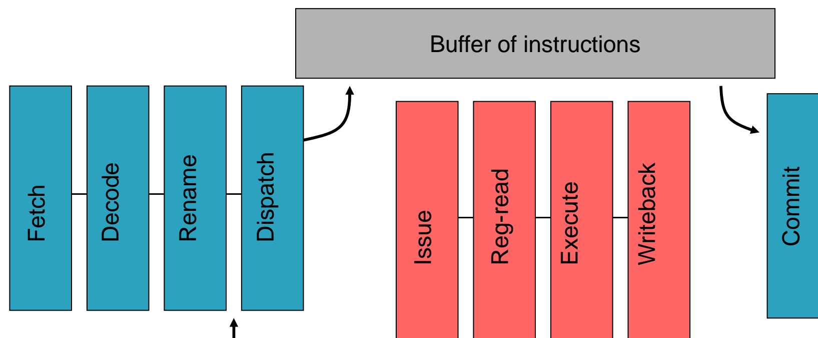
r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6
p1

Free-list

Out-of-order Pipeline



Out-of-Order: Benefits & Challenges

Dynamic Scheduling Operation (Recap)

- Dynamic scheduling
  - Totally in the hardware (not visible to software)
  - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename registers to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies is harder
- "Commit" instructions in order
  - Anything strange happens before commit, just flush the pipeline
- How much out-of-order? Core i7 "Sandy Bridge":
  - 168-entry reorder buffer, 160 integer registers, 54-entry scheduler

i486 Pipeline

- Fetch
  - Load 16-bytes of instruction into prefetch buffer
- Decode1
  - Determine instruction length, instruction type
- Decode2
  - Compute memory address
  - Generate immediate operands
- Execute
  - Register Read
  - ALU operation
  - Memory read/write
- Write-Back
  - Update register file

Pipeline Stage Details

- Fetch
  - Moves 16 bytes of instruction stream into code queue
  - Not required every time
    - About 5 instructions fetched at once
    - Only useful if don't branch
  - Avoids need for separate instruction cache
- D1
  - Determine total instruction length
    - Signals code queue aligner where next instruction begins
  - May require two cycles
    - When multiple operands must be decoded
    - About 6% of "typical" DOS program

Stage Details (Cont.)

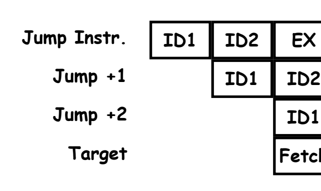
- D2
  - Extract memory displacements and immediate operands
  - Compute memory addresses
    - Add base register, and possibly scaled index register
  - May require two cycles
    - If index register involved, or both address & immediate operand
    - Approx. 5% of executed instructions
- EX
  - Read register operands
  - Compute ALU function
  - Read or write memory (data cache)
- WB
  - Update register result

Data Hazards

- Data Hazards

Generated	Used	Handling
ALU	ALU	EX-EX Forwarding
Load	ALU	EX-EX Forwarding
ALU	Store	EX-EX Forwarding
ALU	Eff. Address	(Stall) + EX-ID2 Forwarding

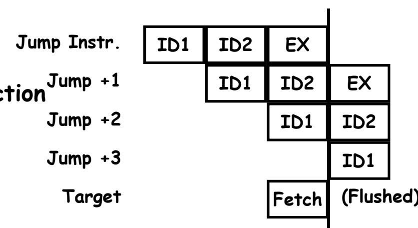
Control Hazards



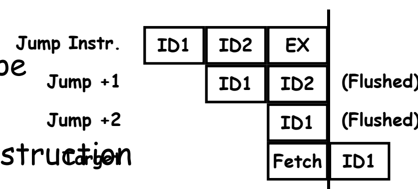
- Jump Instruction Processsing
  - Continue pipeline assuming branch not taken
  - Resolve branch condition in EX stage
  - Also speculatively fetch at target during EX stage

Control Hazards (Cont.)

Branch Not Taken
- **Allow pipeline to continue.**
- **Total of 1 cycle for instruction**



- Branch taken
  - Flush instructions in pipe
  - Begin ID1 at target.
  - Total of 3 cycles for instruction



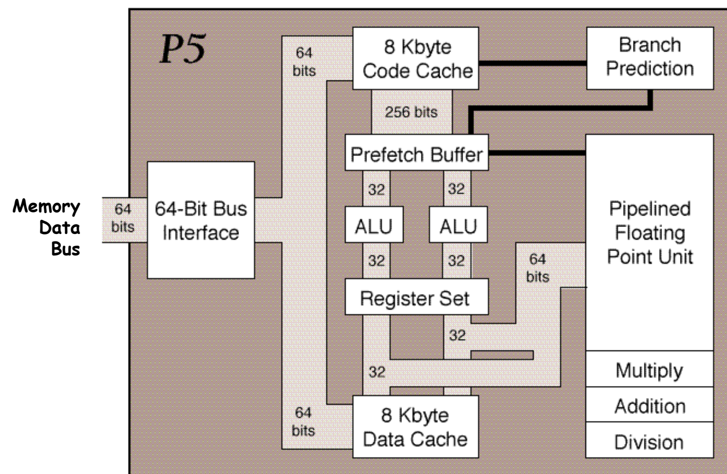
Comparison to 386

- Cycles Per Instruction

Instruction Type	386 Cycles	486 Cycles
Load	4	1
Store	2	1
ALU	2	1
Jump taken	9	3
Jump not taken	3	1
Call	9	3

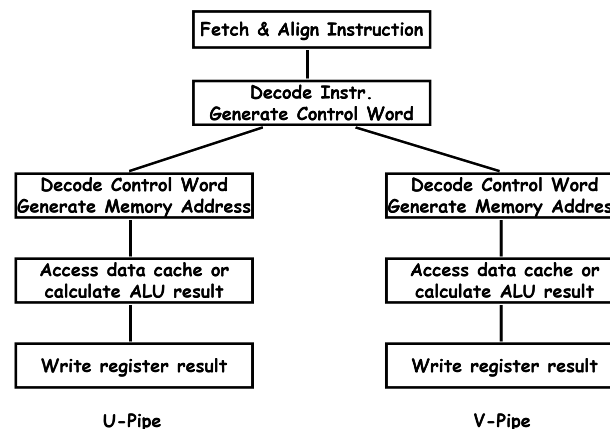
- Reasons for Improvement
  - On chip cache
    - –Faster loads & stores
  - More pipelining

Pentium Block Diagram



(Microcprocessor Report 10/28/92)

Pentium Pipeline



Pentium Pro (P6)

- History
  - Announced in Feb. '95
  - Delivering in high end machines now
- Features
  - Dynamically translates instructions to more regular format
    - Very wide RISC instructions
  - Executes operations in parallel
    - Up to 5 at once
  - Very deep pipeline
    - 12–18 cycle latency

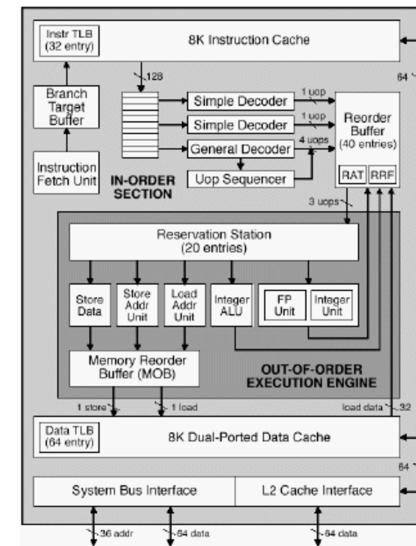
Superscalar Execution

- Can Execute Instructions I1 & I2 in Parallel if:
  - Both are "simple" instructions
    - Don't require microcode sequencing
    - Some operations require U-pipe resources
    - 90% of SpecInt instructions
  - I1 is not a jump
  - Destination of I1 not source of I2
    - But can handle I1 setting CC and I2 being cond. jump
  - Destination of I1 not destination of I2
- If Conditions Don't Hold
  - Issue I1 to U Pipe
  - I2 issued on next cycle
    - Possibly paired with following instruction

Branch Prediction

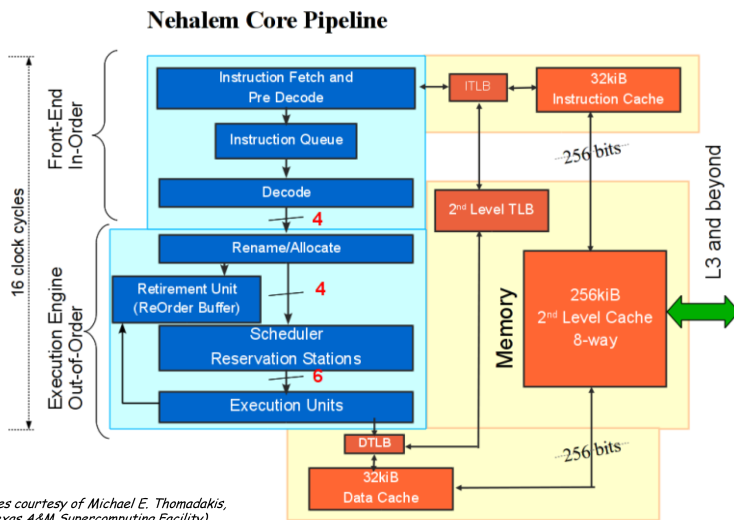
- Branch Target Buffer
  - Stores information about previously executed branches
    - –Indexed by instruction address
    - –Specifies branch destination + whether or not taken
  - 256 entries
- Branch Processing
  - Look for instruction in BTB
  - If found, start fetching at destination
  - Branch condition resolved early in WB
    - –If prediction correct, no branch penalty
    - –If prediction incorrect, lose ~3 cycles
      - » Which corresponds to > 3 instructions
  - Update BTB

PentiumPro Block Diagram



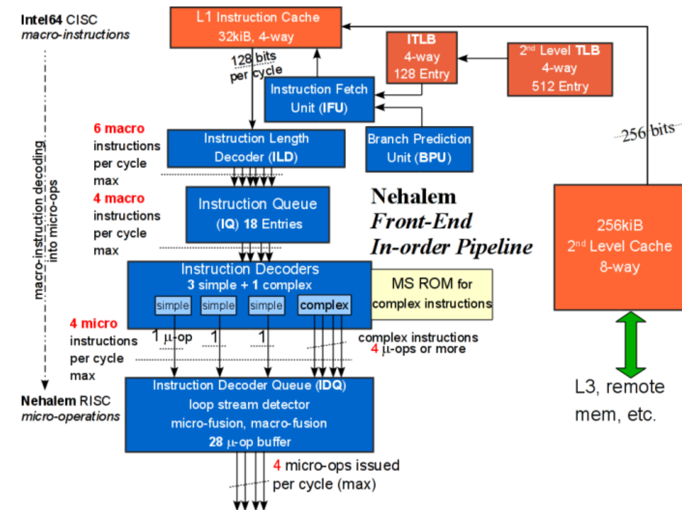
Microprocessor Report 2/16/95

Core i7 Pipeline: Big Picture

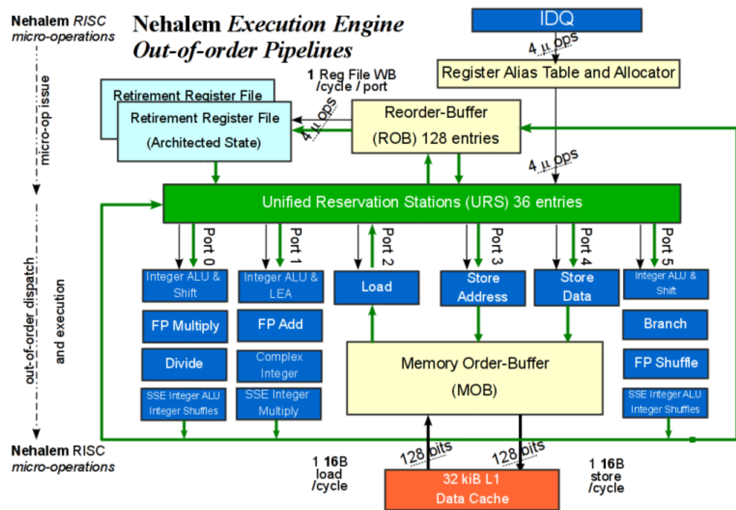


(Images courtesy of Michael E. Thomadakis, Texas A&M Supercomputing Facility)

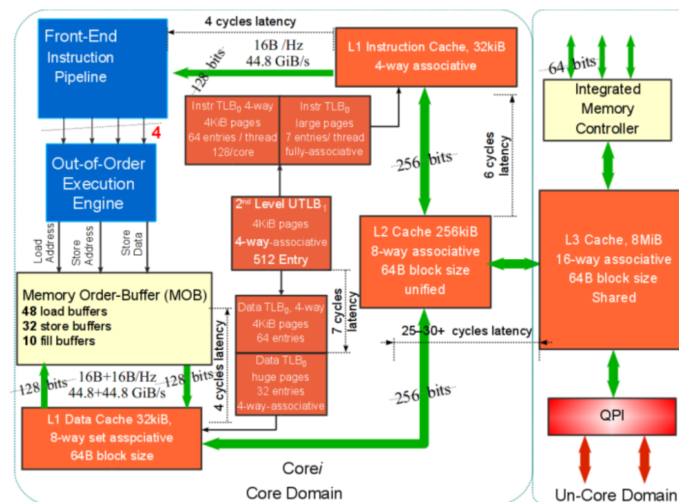
Core i7 Pipeline: Front End



Core i7 Pipeline: Execution Unit



Core i7 Pipeline: Memory Hierarchy



Haswell Buffer Sizes

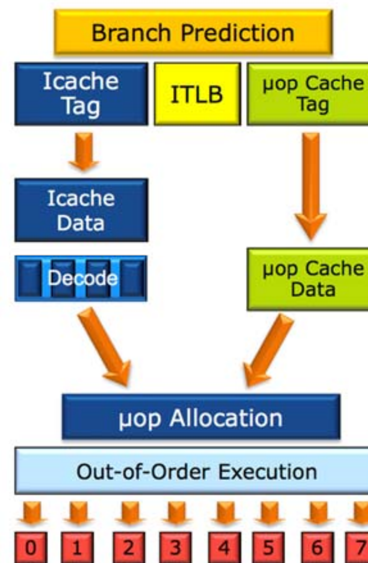
Extract more parallelism in every generation

	Nehalem	Sandy Bridge	Haswell
Out-of-order Window	128	168	192
In-flight Loads	48	64	72
In-flight Stores	32	36	42
Scheduler Entries	36	54	60
Integer Register File	N/A	160	168
FP Register File	N/A	144	168
Allocation Queue	28/thread	28/thread	56

IDF2012 INTEL DEVELOPER FORUM

 Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Nehalem); Intel® Microarchitecture (Sandy Bridge)

Haswell Core at a Glance



Next generation branch prediction
- Improves performance *and* saves wasted work

Improved front-end
- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

Deeper buffers
- Extract more instruction parallelism
- More resources when running a single thread

More execution units, shorter latencies
- Power down when not in use

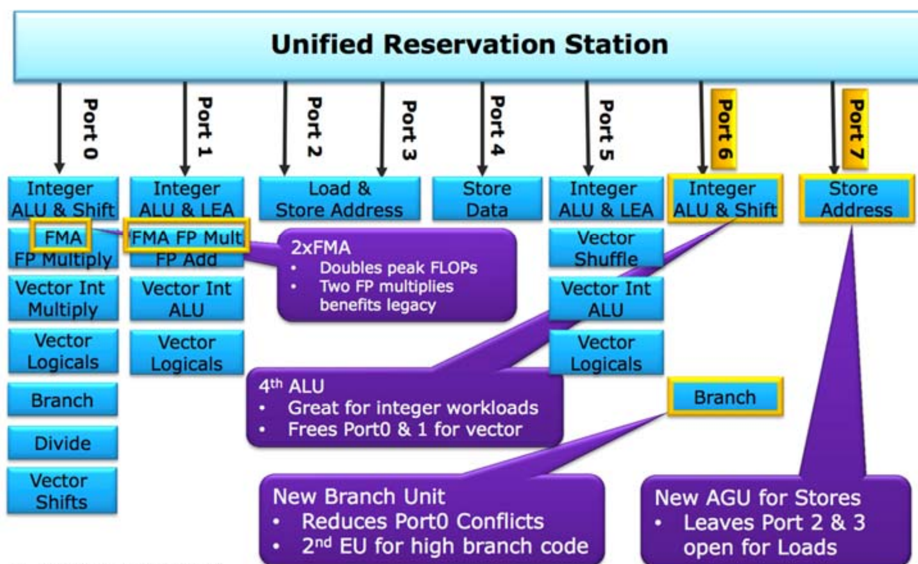
More load/store bandwidth
- Better prefetching, better cache line split latency & throughput, double L2 bandwidth
- New modes save power without losing performance

No pipeline growth
- Same branch misprediction latency
- Same L1/L2 cache latency

IDF2012 INTEL DEVELOPER FORUM

 Intel® Microarchitecture (Haswell)

Haswell Execution Unit Overview



Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way

All caches use 64-byte lines

15 Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

OoO Execution is all around us

- Qualcomm Krait processor (in phones)
  - based on ARM Cortex A15 processor
  - **out-of-order** 1.5GHz dual-core
  - 3-wide fetch/decode
  - 4-wide issue
  - 11-stage integer pipeline
  - 28nm process technology

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction to look past branches
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with
- Schedule can change due to cache misses
- **Memory-level parallelism**
  - Executes "around" cache misses to find independent instructions
  - Finds and initiates independent misses, reducing memory latency
    - Especially good at hiding L2 hits (~12 cycles in Core i7)

Challenges for Out-of-Order Cores

- Design complexity
  - More complicated than in-order? Certainly!
  - But, we have managed to overcome the design complexity
- Clock frequency
  - Can we build a "high ILP" machine at high clock frequency?
  - Yep, with some additional pipe stages, clever design
- Limits to (efficiently) scaling the window and ILP
  - Large physical register file
  - Fast register renaming/wakeup/select/load queue/store queue
    - Active areas of micro-architectural research
  - Branch & memory depend. prediction (limits effective window size)
    - 95% branch mis-prediction: 1 in 20 branches, or 1 in 100 insn.
  - Plus all the issues of building "wide" in-order superscalar
- Power efficiency
  - Today, even mobile phone chips are out-of-order cores

Architectural Performance

- Metric
  - SpecX92/Mhz: Normalizes with respect to clock speed
- Sampling

Processor	MHz	SpecInt92	IntAP	SpecFP92	FltAP
i386/387	33	6	0.2	3	0.1
i486DX	50	28	0.6	13	0.3
Pentium	150	181	1.2	125	0.8
PentiumPro	200	320	1.6	283	1.4
MIPS R3000A	25	16.1	0.6	21.7	0.9
MIPS R10000	200	300	1.5	600	3.0
Alpha 21164a	417	500	1.2	750	1.8

i486 Pipeline

- Fetch
  - Load 16-bytes of instruction into prefetch buffer
- Decode1
  - Determine instruction length, instruction type
- Decode2
  - Compute memory address
  - Generate immediate operands
- Execute
  - Register Read
  - ALU operation
  - Memory read/write
- Write-Back

Pipeline Stage Details

- Fetch
  - Moves 16 bytes of instruction stream into code queue
  - Not required every time
    - About 5 instructions fetched at once
    - Only useful if don't branch
  - Avoids need for separate instruction cache
- D1
  - Determine total instruction length
    - Signals code queue aligner where next instruction begins
  - May require two cycles
    - When multiple operands must be decoded
    - About 6% of "typical" DOS program

Stage Details (Cont.)

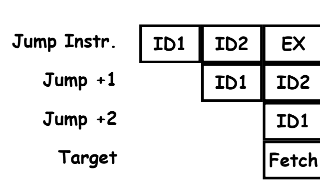
- D2
  - Extract memory displacements and immediate operands
  - Compute memory addresses
    - Add base register, and possibly scaled index register
  - May require two cycles
    - If index register involved, or both address & immediate operand
    - Approx. 5% of executed instructions
- EX
  - Read register operands
  - Compute ALU function
  - Read or write memory (data cache)

Data Hazards

- Data Hazards

Generated	Used	Handling
ALU	ALU	EX-EX Forwarding
Load	ALU	EX-EX Forwarding
ALU	Store	EX-EX Forwarding
ALU	Eff. Address	(Stall) + EX-ID2 Forwarding

Control Hazards

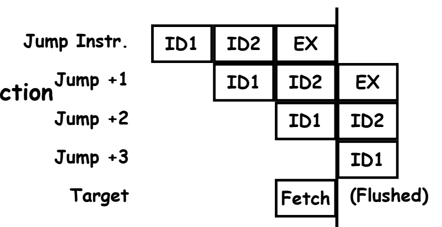


- Jump Instruction Processsing
  - Continue pipeline assuming branch not taken
  - Resolve branch condition in EX stage
  - Also speculatively fetch at target during EX stage

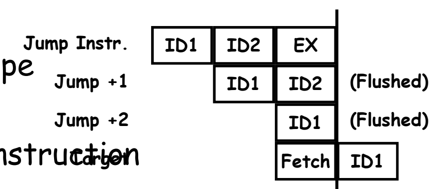
Control Hazards (Cont.)

Branch Not Taken

- Allow pipeline to continue.
- Total of 1 cycle for instruction



- Branch taken
  - Flush instructions in pipe
  - Begin ID1 at target.
  - Total of 3 cycles for instruction



Comparison with Our pAlpha Pipeline

- Two Decoding Stages
  - Harder to decode CISC instructions
  - Effective address calculation in D2
- Multicycle Decoding Stages
  - For more difficult decodings
  - Stalls incoming instructions
- Combined Mem/EX Stage
  - Avoids load stall without load delay slot
    - But introduces stall for address computation

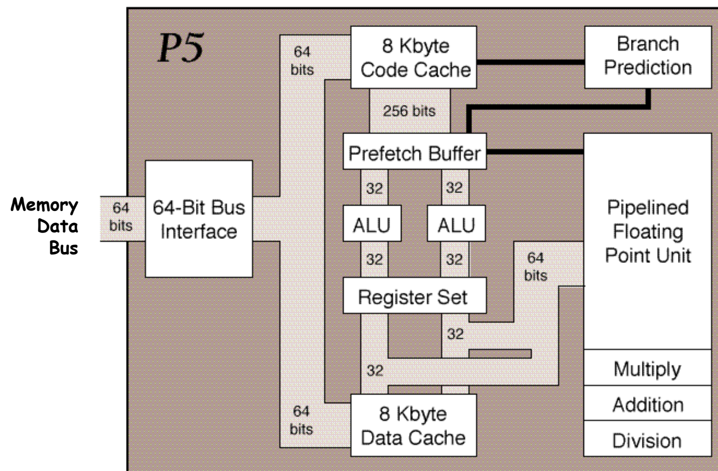
Comparison to 386

- Cycles Per Instruction

Instruction Type	386 Cycles	486 Cycles
Load	4	1
Store	2	1
ALU	2	1
Jump taken	9	3
Jump not taken	3	1
Call	9	3

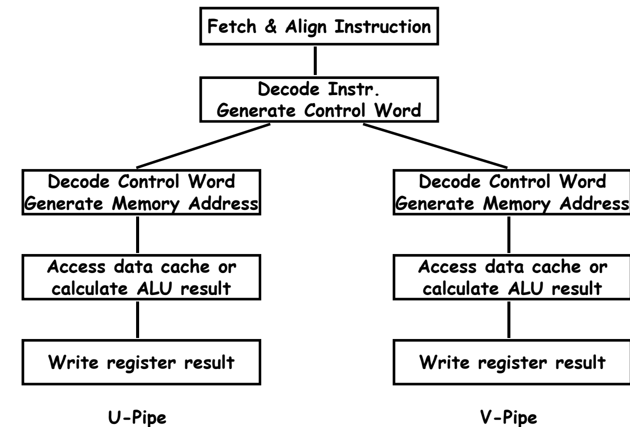
- Reasons for Improvement
  - On chip cache
    - Faster loads & stores
  - More pipelining

Pentium Block Diagram



(Microcprocessor Report 10/28/92)

Pentium Pipeline



Superscalar Execution

- Can Execute Instructions I1 & I2 in Parallel if:
  - Both are "simple" instructions
    - –Don't require microcode sequencing
    - –Some operations require U-pipe resources
    - –90% of SpecInt instructions
  - I1 is not a jump
  - Destination of I1 not source of I2
    - –But can handle I1 setting CC and I2 being cond. jump
  - Destination of I1 not destination of I2
- If Conditions Don't Hold
  - Issue I1 to U Pipe
  - I2 issued on next cycle
    - –Possibly paired with following instruction

Branch Prediction

- Branch Target Buffer
  - Stores information about previously executed branches
    - –Indexed by instruction address
    - –Specifies branch destination + whether or not taken
  - 256 entries
- Branch Processing
  - Look for instruction in BTB
  - If found, start fetching at destination
  - Branch condition resolved early in WB
    - –If prediction correct, no branch penalty
    - –If prediction incorrect, lose ~3 cycles
      - » Which corresponds to > 3 instructions

Superscalar Terminology

- Basic

Superscalar Able to issue > 1 instruction / cycle

Superpipelined Deep, but not superscalar pipeline. E.g., MIPS R5000 has 8 stages

Branch prediction Logic to guess whether or not branch will be taken, and possibly branch target

- Advanced

Out-of-order Able to issue instructions out of program order

Speculation Execute instructions beyond branch points, possibly nullifying later

Register renaming Able to dynamically assign physical registers to instructions

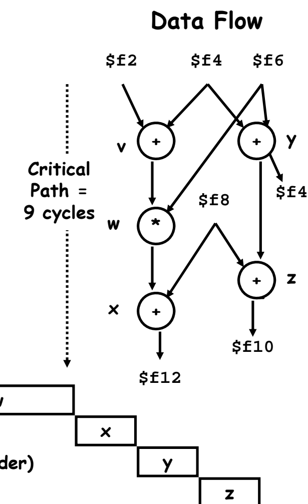
Retire unit Logic to keep track of instructions as

Superscalar Execution Example

- Assumptions
  - Single FP adder takes 2 cycles
  - Single FP multipler takes 5 cycles
  - Can issue add & multiply together
  - Must issue in-order

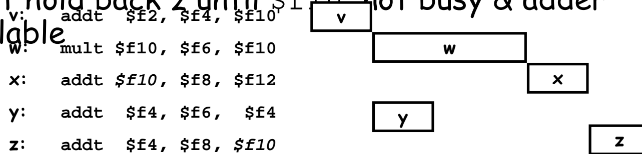
(Single adder, data dependence)
(inorder)

```
v:   addt  $f2, $f4, $f10
w:   mult $f10, $f6, $f10
x:   addt $f10, $f8, $f12
y:   addt  $f4, $f6,  $f4
z:   addt  $f4, $f8, $f10
```

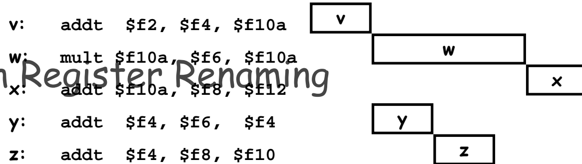


Adding Advanced Features

- Out Of Order Issue
  - Can start y as soon as adder available
  - Must hold back z until $f10 not busy & adder available



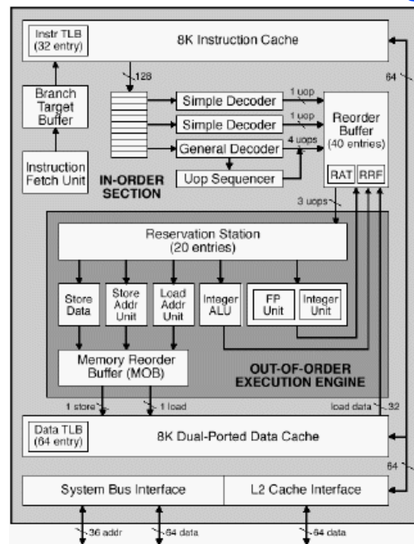
- With Register Renaming



Pentium Pro (P6)

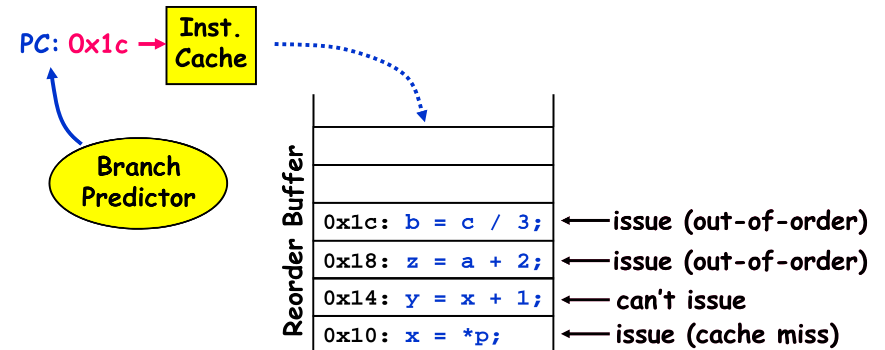
- History
  - Announced in Feb. '95
  - Delivering in high end machines now
- Features
  - Dynamically translates instructions to more regular format
    - Very wide RISC instructions
  - Executes operations in parallel
    - Up to 5 at once
  - Very deep pipeline
    - 12–18 cycle latency

PentiumPro Block Diagram



Microprocessor Report
2/16/95

Dynamically Scheduled Processors



- Fetch & graduate in-order, issue out-of-order

PentiumPro Operation

- Translates instructions dynamically into "Uops"
  - 118 bits wide
  - Holds operation, two sources, and destination
- Executes Uops with "Out of Order" engine
  - Uop executed when
    - Operands available
    - Functional unit available
  - Execution controlled by "Reservation Stations"
    - Keeps track of data dependencies between uops
    - Allocates resources

Read-after-Write (RAW) Dependences

- Also known as a "true" dependence
- Example:

```
S1: addq r1, r2, r3
S2: addq r3, r4, r4
```

- How to optimize?
  - cannot be optimized away

Write-after-Read (WAR) Dependences

- Also known as an "anti" dependence
- Example:

```
S1: addq r1, r2, r3
S2: addq r4, r5, r1
        ...
    addq r1, r6, r7
```

- How to optimize?
  - rename dependent register (e.g., r1 in S2 -> r8)

```
S1: addq r1, r2, r3
S2: addq r4, r5, r8
        ...
    addq r8, r6, r7
```


Write-after-Write (WAW) Dependences

- Also known as an "output" dependence
- Example:

```
S1: addq r1, r2, r3
S2: addq r4, r5, r3
        ...
    addq r3, r6, r7
```

- How to optimize?
  - rename dependent register (e.g., r3 in S2 -> r8)

```
S1: addq r1, r2, r3
S2: addq r4, r5, r8
        ...
    addq r8, r6, r7
```

Living with Expensive Branches

- Mispredicted Branch Carries a High Cost
  - Must flush many in-flight instructions
  - Start fetching at correct target
  - Will get worse with deeper and wider pipelines
- Impact on Programmer / Compiler
  - Avoid conditionals when possible
    - Bit manipulation tricks
  - Use special conditional-move instructions
    - Recent additions to many instruction sets
  - Make branches predictable
    - Very low overhead when predicted correctly

Branch Prediction Example

```
static void loop1() {
    int i;
    data_t abs_sum = (data_t) 0;
    data_t prod = (data_t) 1;
    for (i = 0; i < CNT; i++) {
        data_t x = dat[i];
        data_t ax;
        ax = ABS(x);
        abs_sum += ax;
        prod *= x;
    }
    answer =  abs_sum+prod;
}
```

```
#define ABS(x) x < 0 ? -x : x
```

MIPS Code

```
0x6c4:   8c620000 lw      r2,0(r3)
0x6c8:   24840001 addiu   r4,r4,1
0x6cc:   04410002 bgez    r2,0x6d8
0x6d0:   00a20018 mult    r5,r2
0x6d4:   00021023 subu    r2,r0,r2
0x6d8:   00002812 mflo    r5
0x6dc:   00c23021 addu    r6,r6,r2
0x6e0:   28820400 slti    r2,r4,1024
0x6e4:   1440fff7 bne     r2,r0,0x6c4
0x6e8:   24630004 addiu   r3,r3,4
```

- Compute sum of absolute values
- Compute product of original values

Some Interesting Patterns

- PPPPPPPPP
- +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 …
  - Should give perfect prediction
- RRRRRRRRR
- -1 -1 +1 +1 +1 +1 -1 +1 -1 -1 +1 +1 -1 -1 +1 +1 +1 +1 +1 -1 -1 -1 +1 -1 …
  - Will mispredict 1/2 of the time
- N*N[PNPN]
- -1 -1 -1 -1 -1 -1 -1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 …
  - Should alternate between states No! and No?
- N*P[PNPN]
- -1 -1 -1 -1 -1 -1 -1 +1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 …
  - Should alternate between states No? and Yes?
- N*N[PPNN]
- -1 -1 -1 -1 -1 -1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 +1 +1 -1 -1 …

Loop Performance (FP)

	R3000		PPC 604		Pentium	
Pattern	Cycles	Penalty	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	13.6	0	9.2	0	21.1	0
RRRRRRRRR	13.6	0	12.6	3.4	22.9	1.8
N*N[PNPN]	13.6	0	15.8	6.6	23.3	2.2
N*P[PNPN]	13.3	-0.3	15.9	6.7	24.3	3.2
N*N[PPNN]	13.3	-0.3	12.5	3.3	23.9	2.8
N*P[PPNN]	13.6	0	12.5	3.3	24.7	3.6

- Observations
  - 604 has prediction rates 0%, 50%, and 100%
    - Expected 50% from N*N[PNPN]
    - Expected 25% from N*N[PPNN]
    - Loop so tight that speculate through single branch twice?
  - Pentium appears to be more variable, ranging 0 to 100%

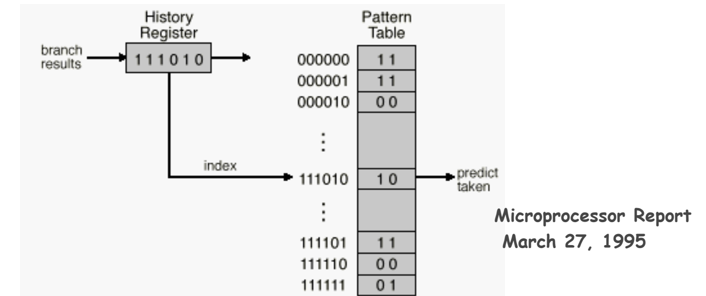
Loop 1 Surprises

	R10000		Pentium II	
Pattern	Cycles	Penalty	Cycles	Penalty
PPPPPPPPP	3.5	0	11.9	0
RRRRRRRRR	3.5	0	19	7.1
N*N[PNPN]	3.5	0	12.5	0.6
N*P[PNPN]	3.5	0	13	1.1
N*N[PPNN]	3.5	0	12.4	0.5
N*P[PPNN]	3.5	0	12.2	0.3

- Pentium II
  - Random shows clear penalty
  - But others do well
    - More clever prediction algorithm
- R10000
  - Has special "conditional move" instructions
  - Compiler translates `a = Cond ? Texpr : Fexpr` into

```
a = Fexpr
temp = Texpr
```

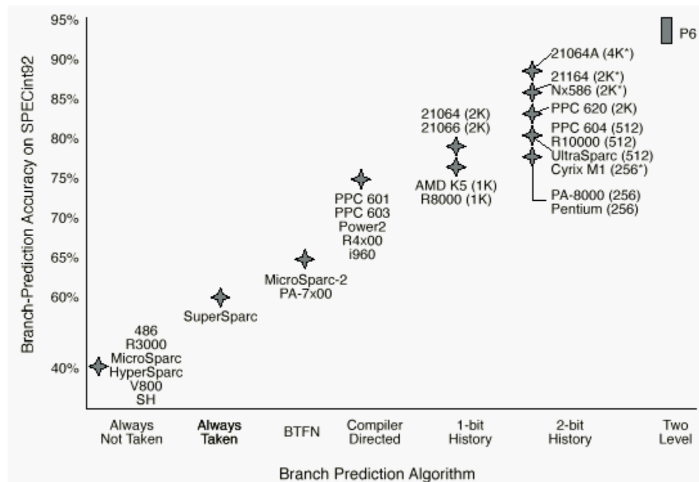

P6 Branch Prediction



Microprocessor Report March 27, 1995

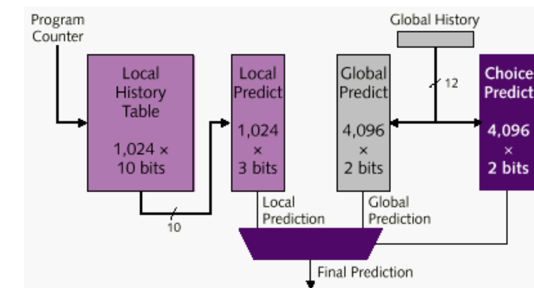
- Two-Level Scheme
  - Yeh & Patt, ISCA '93
  - Keep shift register showing past k outcomes for branch
  - Use to index 2^k entry table
  - Each entry provides 2-bit, saturating counter

Branch Prediction Comparisons



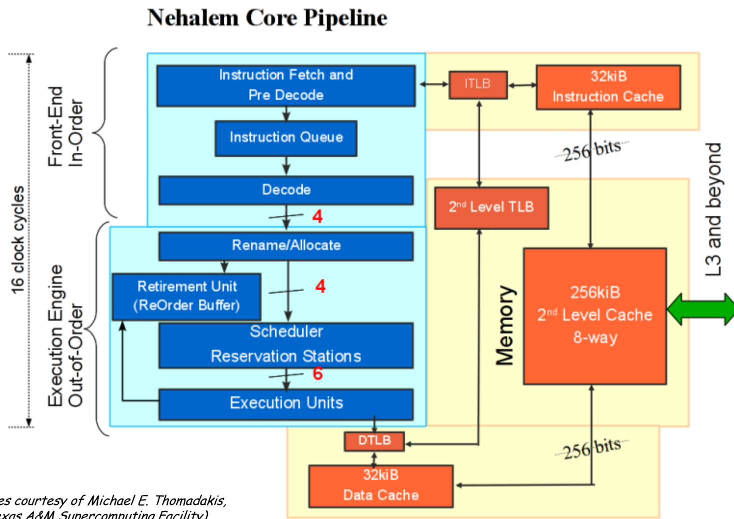
- Microprocessor Report March 27, 1995

21264 Branch Prediction Logic



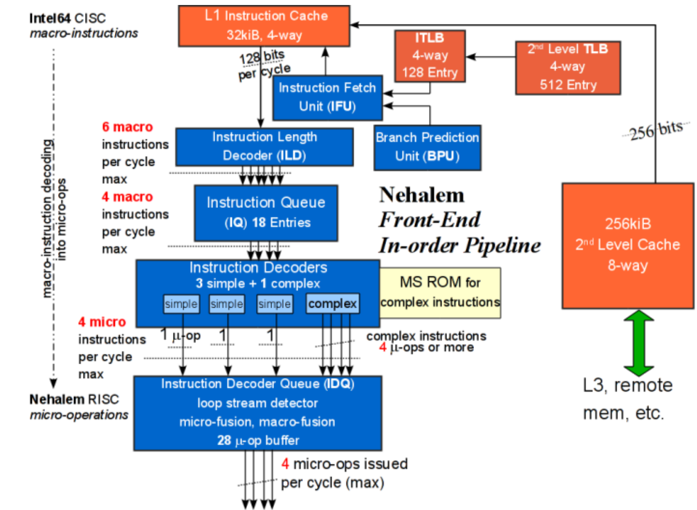
- Purpose: Predict whether or not branch taken
- 35Kb of prediction information
- 2% of total die size
- Claim 0.7--1.0% misprediction

Core i7 Pipeline: Big Picture

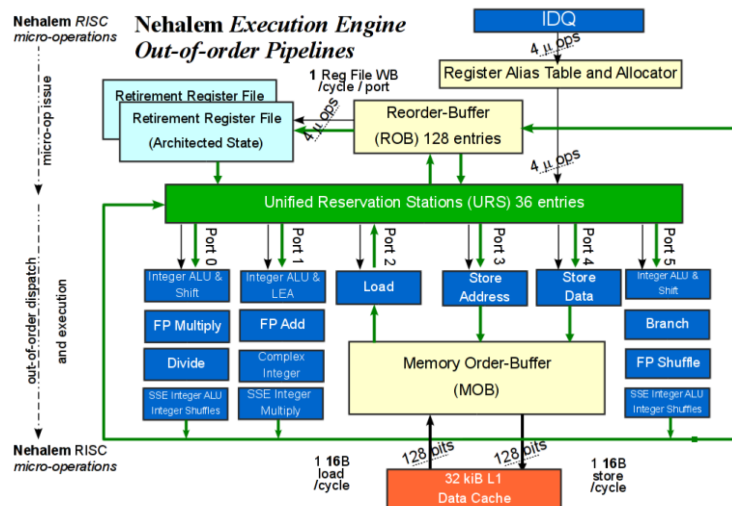


(Images courtesy of Michael E. Thomadakis, Texas A&M Supercomputing Facility)

Core i7 Pipeline: Front End



Core i7 Pipeline: Execution Unit



Core i7 Pipeline: Memory Hierarchy