

Superscalar Architectures

Mark Greenstreet

CpSc 418 – Oct. 2, 2012

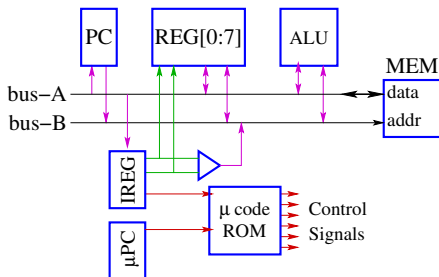
Outline:

- Computer Architecture Overview
 - ▶ Computers from before when the instructor was born (i.e. **really** old).
 - ▶ Microcoded machines (like you saw in CpSc 121).
 - ▶ Pipelined machines (like you saw in CpSc 313).
- Superscalar architectures
- Matrix multiplication on a superscalar machine.

In the beginning. . .

- “Computer” was the name for a profession.
 - ▶ problems that required large numbers of calculations were solved by rooms full of people.
- Then, we got mechanical calculators.
- The first electronic computers were not programmable.
 - ▶ The various functional units were connected together with “patch cords” like an old-fashioned telephone exchange.
 - ▶ The compute would be configured for a particular problem.
 - ▶ It would run to solve the problem,
 - ▶ And then it would be reconfigured for the next problem.
- The use of memory for program and data started was proposed after WW II.
- Each machine had it’s own machine language to match its functional units.
 - ▶ Software had to be rewritten for each different machine.

Microcoded machines



A simple, microcoded machine

- The microcode (μ code) ROM specifies the sequence of operations necessary to carry out an instruction.
- For simplicity, I'm assuming that the op-code bits of the instruction form the most significant bits of the μ code ROM address, and that the value of the micro-PC (μ PC) form the lower half of the address.

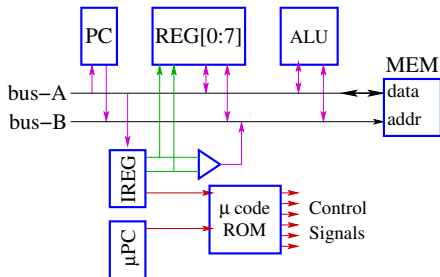
Microcode: example

Example: `add reg[2], reg[3]`

$\mu\text{PC} = 0$: `bus-A` \leftarrow `reg[2]`,
`bus-B` \leftarrow `reg[3]`,
`ALU-op` \leftarrow *add*;

$\mu\text{PC} = 1$: `bus-A` \leftarrow `ALU`,
`reg[2]` \leftarrow `bus-A`
`PC-op` \leftarrow *increment*

$\mu\text{PC} = 2$: `bus-B` \leftarrow `PC`,
`mem-op` \leftarrow *read*
`IREG` \leftarrow `bus-A`
 $\mu\text{PC-op}$ \leftarrow *reset*



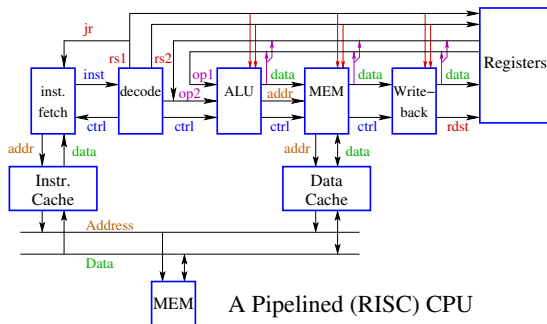
Microcode: summary

- Separates hardware from instruction set.
 - ▶ Different hardware can run the same software.
 - ▶ Enabled IBM to sell machines with a wide range of performance that were all compatible
 - ★ I.e. IBM built an empire and made a fortune on the IBM 360 and its successors.
 - ★ Intel has done the same with the x86.
- **But**, as implemented on slide 3, it's **very** sequential.

```
while(true) {  
    fetch an instruction;  
    perform the instruction  
}
```

- Instruction fetch is “overhead”
 - ▶ Motivates coming up with complicated instructions that perform lots of operations per instruction fetch.
 - ▶ But these are hard for compilers to use.
 - ▶ Can we do better?

Pipelined instruction execution



A Pipelined (RISC) CPU

- Successive instructions in each stage
- When instruction i in *ifetch*, instruction $i-1$ in *decode*, ...
- Allows throughput of one instruction per cycle.
- Favors simple instructions that execute on a single pass through the pipeline.

Execution Example: matrix multiplication

$$\begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} * \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

$$c_{1,1} = a_{1,1} * b_{1,1} + a_{1,2} * b_{2,1}$$

$$c_{1,2} = a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2}$$

$$c_{2,1} = a_{2,1} * b_{1,1} + a_{2,2} * b_{2,1}$$

$$c_{2,2} = a_{2,1} * b_{1,2} + a_{2,2} * b_{2,2}$$

On the next slide

- Register `$r1` points to the base of array *A*.
- Register `$r2` points to the base of array *B*.
- Register `$r3` points to the base of array *C*.
- Registers `$f0` to `$f3` will hold the elements of *A*.
- Registers `$f4` to `$f7` will hold the elements of *B*.
- Registers `$f8` to `$f11` will hold the elements of *C*.
- Register `$f12` is used as a temporary.

Matrix Multiply in (Simplified MIPS) Assembly Code

```
PC   OP      dsd      src
0100 ld      $f0 ← 0($r1)    %  $a_{1,1}$ 
0104 ld      $f4 ← 0($r2)    %  $b_{1,1}$ 
0108 mult    $f8 ← $f0, $f4  %  $a_{1,1} * b_{1,1}$ 
010c ld      $f1 ← 8($r1)    %  $a_{1,2}$ 
0110 ld      $f6 ← 16($r2)   %  $b_{2,1}$ 
0114 mult    $f12 ← $f1, $f6 %  $a_{1,2} * b_{2,1}$ 
0118 add     $f8 ← $f8, $f12 %  $c_{1,1}$ 
011c ld      $f5 ← 8($r2)    %  $b_{1,2}$ 
0120 mult    $f9 ← $f0, $f5  %  $a_{1,1} * b_{1,2}$ 
0124 ld      $f7 ← 24($r2)   %  $b_{2,2}$ 
0128 mult    $f12 ← $f1, $f7 %  $a_{1,2} * b_{2,2}$ 
012c add     $f9 ← $f9, $f12 %  $c_{1,2}$ 
...

```

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$	•				
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$		•			
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$	•				
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$			•		
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$		•			
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$	•				
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$				•	
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$		•			
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$	•				
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$					
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$					
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$					
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$					•
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					•
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$		•	○		
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$	•				
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$					
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$					
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$					
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions

- ▶ Multiplies and adds take one cycle.
- ▶ Cache access takes one cycle.
- ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$					
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					•
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$			•	○	
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$		•			
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$	•				
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$					
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$					
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$				•	○
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$	•				
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$		•			
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					•
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$				•	
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$			•		
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$		•			
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$	•				
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$					
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$					•
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$					•
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$		•	○		
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$	•				
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$					
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$					
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$					
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$					•
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$			•	○	
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$		•			
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$	•				
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$				•	○
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$				•	
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$	•				
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					•
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$				•	
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$			•		
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$		•			
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$	•				
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					•
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					•
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$		•	○		
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$	•				
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					•
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$			•	○	
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$			•		
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$	•				
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$				•	○
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$				•	
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$			•		
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$	•				
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					•
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					•
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					•
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$	•		•	○	
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					•
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$				•	○
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$	•				
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$				•	○
...								•	

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	← 0(\$r1)	% $a_{1,1}$					
0104	ld	\$f4	← 0(\$r2)	% $b_{1,1}$					
0108	mult	\$f8	← \$f0, \$f4	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	← 8(\$r1)	% $a_{1,2}$					
0110	ld	\$f6	← 16(\$r2)	% $b_{2,1}$					
0114	mult	\$f12	← \$f1, \$f6	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	← \$f8, \$f12	% $c_{1,1}$					
011c	ld	\$f5	← 8(\$r2)	% $b_{1,2}$					
0120	mult	\$f9	← \$f0, \$f5	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	← 24(\$r2)	% $b_{2,2}$					
0128	mult	\$f12	← \$f1, \$f7	% $a_{1,2} * b_{2,2}$					•
012c	add	\$f9	← \$f9, \$f12	% $c_{1,2}$				•	
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Matrix Multiply in (Simplified MIPS) Assembly Code

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0104	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
010c	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0110	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
011c	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0124	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0128	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,2}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					
...									

- Simplifying/optimistic assumptions
 - ▶ Multiplies and adds take one cycle.
 - ▶ Cache access takes one cycle.
 - ▶ No cache misses.

Ban the Bubbles

PC	OP	dsd	src	
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$	•				
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$		•			
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$	•				
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$			•		
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$		•			
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$	•				
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					•
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$				•	
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$	•				
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src	
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$

IF DEC ALU MEM WB

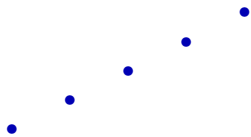


- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src	
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$

IF DEC ALU MEM WB



- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src	
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$

IF DEC ALU MEM WB



- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src	
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$

IF DEC ALU MEM WB



- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					•
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					•
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					•
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					•
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$	•				
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$		•			
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					•
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$		•			
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$	•				
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					•
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					•
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$				•	
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$				•	
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$	•				
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					•
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					•
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$				•	
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$				•	
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$	•				
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					•
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$				•	
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$			•		
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$		•			
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$	•				
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					•
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					•
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$				•	
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$			•		
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$	•				

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					•
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$				•	
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$			•		
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$		•			

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					•
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$				•	
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$				•	

- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					•
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$				•	

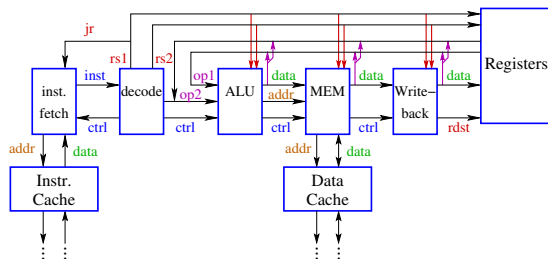
- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Ban the Bubbles

PC	OP	dsd	src		IF	DEC	ALU	MEM	WB
0100	ld	\$f0	$\leftarrow 0(\$r1)$	% $a_{1,1}$					
0100	ld	\$f1	$\leftarrow 8(\$r1)$	% $a_{1,2}$					
0100	ld	\$f2	$\leftarrow 16(\$r1)$	% $a_{2,1}$					
0100	ld	\$f3	$\leftarrow 24(\$r1)$	% $a_{2,2}$					
0100	ld	\$f4	$\leftarrow 0(\$r2)$	% $b_{1,1}$					
0100	ld	\$f5	$\leftarrow 8(\$r2)$	% $b_{1,2}$					
0100	ld	\$f6	$\leftarrow 16(\$r2)$	% $b_{2,1}$					
0100	ld	\$f7	$\leftarrow 24(\$r2)$	% $b_{2,2}$					
0108	mult	\$f8	$\leftarrow \$f0, \$f4$	% $a_{1,1} * b_{1,1}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f6$	% $a_{1,2} * b_{2,1}$					
0118	add	\$f8	$\leftarrow \$f8, \$f12$	% $c_{1,1}$					
0120	mult	\$f9	$\leftarrow \$f0, \$f5$	% $a_{1,1} * b_{1,2}$					
0114	mult	\$f12	$\leftarrow \$f1, \$f7$	% $a_{1,2} * b_{2,1}$					
012c	add	\$f9	$\leftarrow \$f9, \$f12$	% $c_{1,2}$					

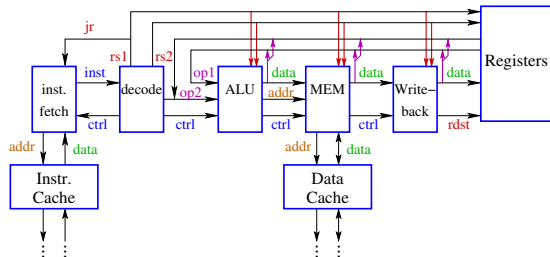
- Load all of A and B .
- No more loads needed to compute $c_{2,1}$ and $c_{2,1}$.

Pipeline Hazards (1/2)



- A hazard occurs if pipelined instruction execution produces a different result than sequential execution.
- Data hazard: instruction i reads register (in **decode** stage) that instruction $i-1$ will write later (in **Write-back**)
 - ▶ Standard solution: bypasses.
 - ▶ Decode broadcasts registers to be fetched to all pipestages:
 - ★ If a stage has an instruction that will write that register.
 - ★ It provides the value or forces a stall.

Pipeline Hazards (2/2)



- Control Hazard: branch not resolved before next instruction fetched.
 - ▶ Standard solutions:
 - ★ Resolve branches early (in decode stage).
 - ★ Expose one-cycle “delay-slot” to compiler/assembler.

Exceptions

An **exception** causes the CPU to switch to executing a different sequence of instructions:

- system calls: trap into the operating system, switch from user to supervisor execution mode.
- page faults: the virtual address for a load or store doesn't have a corresponding physical memory location. Trap into the OS to bring the appropriate page in from disk and update the page tables.
- other exceptions caused by software: illegal addresses, overflow, illegal instruction. . . .
- interrupts = exceptions caused by the hardware
 - ▶ I/O device interrupts (e.g. data ready from disk)
 - ▶ timer interrupts (e.g. the timer for the process scheduler)
 - ▶ hardware failures (e.g. uncorrectable memory read error)
 - ▶ some notebook computers have an accelerometer interrupt: "This laptop is falling, retract the disk heads before impact."

Precise Exceptions

- To restart a process after handling an exception, the OS needs to be able to reconstruct the process state. This also helps to identify the cause of the exception.
- An exception is **precise** if:
 - ▶ The exception is associated with a particular instruction.
 - ▶ All instructions (in program order) prior to the faulting instruction complete execution.
 - ▶ The faulting instruction and all subsequent ones appear to have never started.
- Programmers love precise exceptions, especially systems programmers 😊.
- Precise exceptions are fairly straightforward to implement on an “old-fashioned” CPU:
 - ▶ However, some complex instructions can cause challenges.
 - ▶ E.g. how do you handle a page-fault in the middle of a huge memory-block-copy instruction?

Precise Exceptions on a RISC

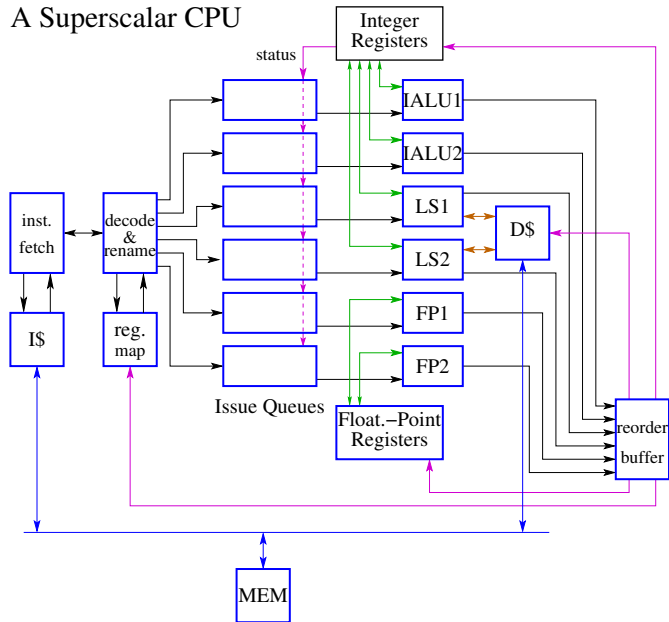
- How to do it:
 - ▶ Let the faulting instruction continue to the write-back stage.
 - ▶ When it reaches write-back, cancel all instructions in previous stages.
 - ★ This means making sure that the following instruction doesn't do a write in the `MEM` stage.
 - ▶ Handle the exception.
 - ▶ The process can resume by restarting at the faulting instruction.
- Exceptions and hazards:
 - ▶ Data hazards: must be handled by bypasses and stalls.
Can't have the result from instruction `i` visible to instruction `i+1` only if instruction `i` faults.
 - ▶ Control hazards: record delay-slot info as part as part of the per-process data in the OS kernel.

Outline

- Computer Architecture Overview
- Superscalar architectures
 - ▶ Multiple-instruction issue
 - ▶ Register renaming
 - ▶ Branches, exceptions, memory accesses
- Matrix multiplication on a superscalar machine.

Superscalar Processors

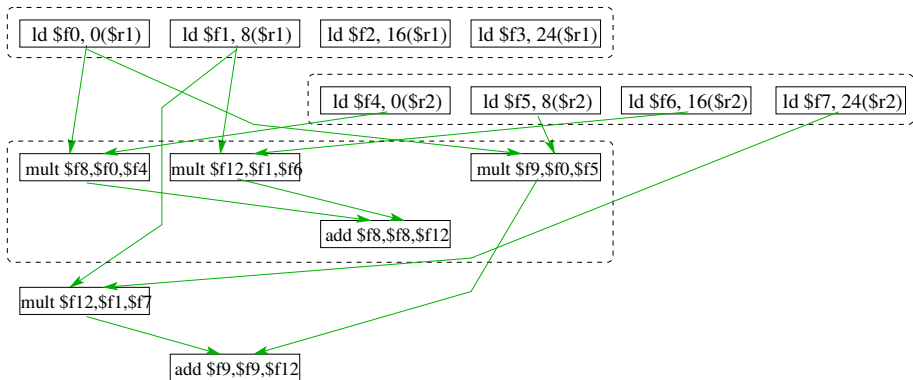
A Superscalar CPU



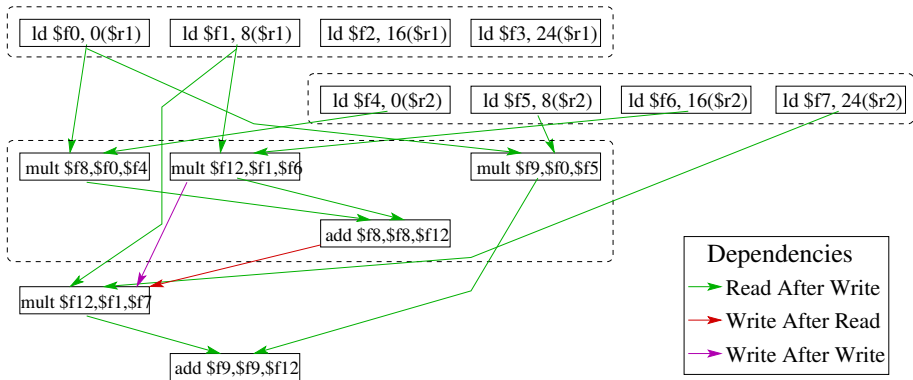
Superscalar Execution

- Fetch several, W , instructions each cycle.
- Decode them in parallel, and send them to issue queues for the appropriate functional unit.
- But what about hazards?
 - ▶ We need to make sure that data and dependencies are properly observed.
 - ▶ Code should execute on a superscalar **as if** it were executing on sequential, one-instruction-at-a-time machine.

Dependencies



Dependencies



- There are also **control dependencies**.

Register Renaming

Like everything else in computer science, superscalars just add another level of indirection to solve the problem of data dependencies.

- Logical and Physical Registers
 - ▶ Machine-code (assembly) instructions refer to **logical** registers.
 - ▶ The machine stores values in **physical** registers.
 - ▶ The machine maintains a mapping between the logical and physical registers.
- Renaming:
 - ▶ When an instruction is decoded,
 - ★ logical registers that it reads are mapped to physical register according to the current register map.
 - ★ for each register that the instruction writes:
 - a physical register is allocated from a free list.
 - the register map is updated to make this new physical register for the logical register written by the instruction.
 - ▶ I'll explain how registers get back to the freelist shortly.

Renaming Example (1/2)

Consider executing:

```
add $5, $2, $3      % reg[5] → reg[2] + reg[3]
st  24($8), $5      % MEM[24+$8] → reg[4]
...o$5, $7, #0x7f  % the next instruction to write reg[5]
```

With

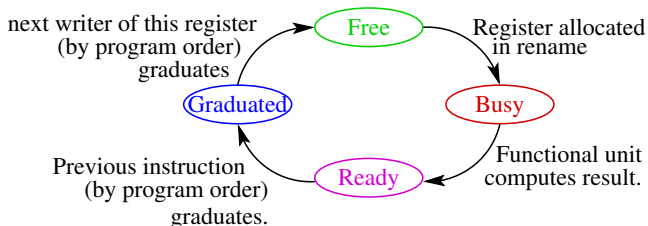
- Physical registers P47 and P23 the next two entries on the freelist.
- The current register map includes:

Logical	→	Physical
2		14
3		61
4		21
5		38
...		

Renaming Example (2/2)

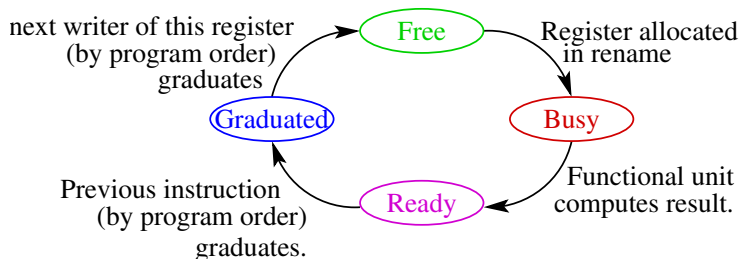
- When `add $5, $2, $3` goes through decode and rename
 - ▶ it becomes `add P47, P14, P61`.
 - ▶ register `P47` is removed from the free list and marked as **busy**.
 - ▶ the “renamed” instruction is added to the issue-queue of an ALU.
- If register `P14` or `P61` are marked as **busy**,
 - ▶ that indicates that the instruction that will set the value for that register hasn’t executed yet.
 - ▶ the `add` instruction will have to wait.
- When both registers `P14` or `P61` are marked as **ready** (or **graduated**), then the `add` instruction can execute.
- When `add P47, P14, P61` executes, register `P47` is marked as **ready**.
- When all instructions that came before `add P47, P14, P61` in program order have **graduated**, then register `P47` graduates as well.
- When the instruction `or $5, $7, #0x7f` graduates, then register `P47` returns to the freelist.

The Lifecycle of a Register (1/2)



- Renaming constructs a graph that reflects the data dependencies of the program.
- It removes “false” hazards such as
 - ▶ Write-after-write:
 - ★ Two instructions that write the same logical register can execute in either order.
 - ★ The last one in program order will be the last one to graduate.
 - ▶ Write-after-read:
 - ★ An instruction that writes a logical register can execute before an earlier instruction that reads the same logical register.
 - ★ That’s because the two occurrences of the logical register map to different physical registers.

The Lifecycle of a Register (1/2)



- Renaming removes “false” hazards such as
 - ▶ Write-after-write.
 - ▶ Write-after-read.
- The set of graduated registers corresponds to the state of a sequential execution of the program after executing the last graduated instruction.

Branch Prediction

- A branch may not be resolved until several cycles after it is fetched.
- Many instructions will have been fetched, decoded, queued and perhaps executed (but not graduated) in the meantime.
 - ▶ How does the processor determine which instructions to fetch?
- Speculate
 - ▶ Keep track of recent history of branches – typically with “saturating counters”.
 - ▶ If the counter predicts branch taken, the fetch unit sees this and fetches from the branch target.
 - ▶ If the counter predicts branch not-taken, the fetch unit continues fetching in sequence.
 - ▶ If a branch is mispredicted
 - ★ Then the **wrong path** instructions are aborted.
 - ★ The processor resumes execution down the correct path.
 - ★ Branch mispredict penalties can be large.

Does Branch Prediction Work?

Good:

- For-loops with many executions: predict branch taken.
- If statements of the form

```
if(condition) {  
    take care of the common case;  
} else {  
    handle unusual situation;  
}
```

– predict branch not taken (i.e. `then` clause will be executed).

Bad:

- Data dependent branches:

```
while((x < xtop) && (y < ytp)) { % merge-loop  
    if(*x < *y) *(z++) = *(x++);  
    else *(z++) = *(y++);  
}
```

- If the data to be sorted is random
 - ▶ Any branch predictor will be wrong 50% of the time.
 - ▶ Even the mispredicted path can help reduce cache-misses.

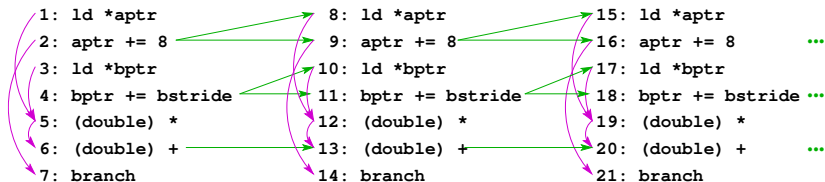
Exceptions on a Superscalar

- Instructions commit in program order \Rightarrow
 - ▶ Take the exception when the instruction would graduate.
- This ensures that all previous instructions have graduated.
- Abort all subsequent instructions.
 - ▶ This ensures that the current instruction and all subsequent ones appear to have never started.
 - ▶ Register restored by unwinding the register mappings to the point they were just before decoding and mapping the faulting instruction.
 - ▶ Memory has the correct values because stores are delayed until the store instruction graduates.

Example: matrix multiply

```
for(int i = 0; i < n_rows_a; i++) {
    for(int j = 0; j < n_cols_b; j++) {
        sum = 0.0; for(int k = 0; k < n_cols_a; k++) {
            % n_cols_a = n_rows_b
            sum += a[i,k] * b[k,j];
        }
        c[i,j] = sum;
    }
}
```

Inner-Loop: Dependencies

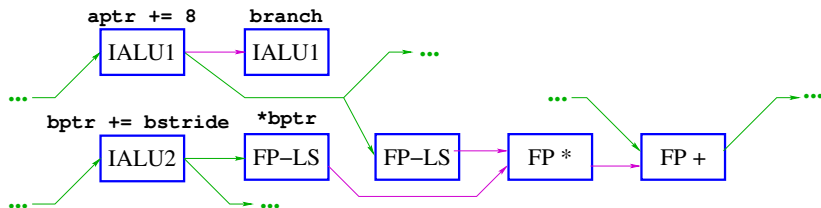


- Number to the left of instructions indicate the program order of instruction execution.
- Magenta arcs indicate data dependencies within a single iteration of the loop body.
- Green arcs indicate data dependencies between loop iterations.

Inner-Loop: Execution

- The pointer increments only depend on the previous loop:
 - ▶ These can “run ahead” of the rest of the instructions, computing addresses well in advance.
 - ▶ With renaming, there will one physical registers per active loop iteration for the `aptr` logical register, and likewise for the `bptr` register.
- The `ld` and `branch` instructions only depend on the values of `aptr` and `bptr` and will execute behind them.
 - ▶ For large matrices, the branch predictor will predict that the branch is always taken.
 - ★ At the end of the loop, the branch operations will be well-ahead of the floating point operations.
 - ★ The mispredicted branch will be detected and the wrong-path instructions aborted well-before the extra floating point operations are actually performed.
 - ▶ If a load misses in the cache,
 - ★ subsequent loads will still issue and be executed.
 - ★ This allows multiple cache misses to be processed in an overlapping manner.

Inner-loop: pipelining



- Thus, the machine acts like the pipeline shown above.
 - ▶ Register renaming ensures that the right values are processed by each instruction.
 - ▶ The critical bottlenecks for a MIPS R10000 are the floating-point load-store unit, `FP-LS`, and the integer ALU, `IALU1`.
 - ★ Each can perform one operation per cycle.
 - ★ Each needs to perform two operations per loop iteration.
 - ★ Thus, the MIPS R10000 can perform one iteration of the inner loop every two cycles.
 - ★ That's seven instructions in two clock cycles for an IPC (instructions-per-cycle) of 3.5.

Super-scalar scaling (1/2)

- For matrix multiply, we got an IPC of 3.5.
 - ▶ That was assuming no cache misses.
 - ▶ As noted on slide 29, the out-of-order execution can help reduce the performance degradation due to cache misses.
 - ▶ In practice, IPC's of ~ 1.5 are typical for super-scalars running desk-top applications.
- Can we get better performance by increasing the parallelism of the superscalar?
 - ▶ Many key structures of the super-scalar have areas that grow quadratically with the issue width.
 - ★ Register-files: both width and height grow linearly with the number of read and write ports.
 - ★ Renaming: to rename W instructions in one cycle,
 - the rename unit must compare the destination register numbers of earlier instructions with the source register numbers of later instructions in the batch.
 - the number of comparators in the rename unit grows quadratically with W .
 - ★ For similar reasons, the size of the reorder buffer (for graduating instructions) must grow quadratically with W .

Super-scalar scaling (2/2)

- In many ways, a super-scalar is a **communication maximizing** architecture.
- It made sense when wires were cheap and functional units were expensive.
- Programmers like it because it finds the implicit parallelism in the machine instructions – no special programming is needed.
- The trend is for multiple cores with smaller issue-width super scalars (i.e. smaller values of W).
- Simplified super-scalars (i.e. fetch two instructions per cycle) may survive for a long time,
 - ▶ Or we may see that simpler, RISC pipelines prevail,
 - ▶ or some other simplified architecture.

Lynn Conway

- Worked on first super-scalar processor design at IBM.
- Was subsequently fired from IBM. Why?
- Worked as a programmer at Memorex for a few years, and then went to Xerox PARC.
- Collaborated with Carver Mead (Caltech) to start the “VLSI revolution”
 - ▶ Key idea was to apply principles of abstract from computer science to integrated circuit design.
 - ▶ footnotesize approach has completely transformed the industry and made large, multi-billion transistor designs possible.
- For more information, see:

<http://ai.eecs.umich.edu/people/conway/>

Photo from

http://en.wikipedia.org/wiki/File:Lynn_Conway_July_2006.jpg



Why does it matter?

- Role models matter
 - ▶ If you're a straight, white or asian, male in computer science,
 - ★ then there are lots of people like you who can be role models.
 - ★ If you're like me, you'll often take this for granted, and not even think of them as role models.
 - ▶ The further you are from this “center-of-mass” of the field,
 - ★ the sparser role models become,
 - ★ and you may feel like you don't fit in.
 - ★ If so, please get the message from this that **you're not the problem.**
- Lynn Conway has lived a remarkable life
 - ▶ She's made important contributions to computer architecture, VLSI design, and robotics.
 - ▶ She's chosen to use her experiences to help others who are facing similar challenges and discrimination.

Announcements

- Oct. 4: Shared Memory Multi-Processors
- Oct. 9: Message Passing Computers
- Oct. 11: Models of Parallel Computation or [DLS Lecture](#).