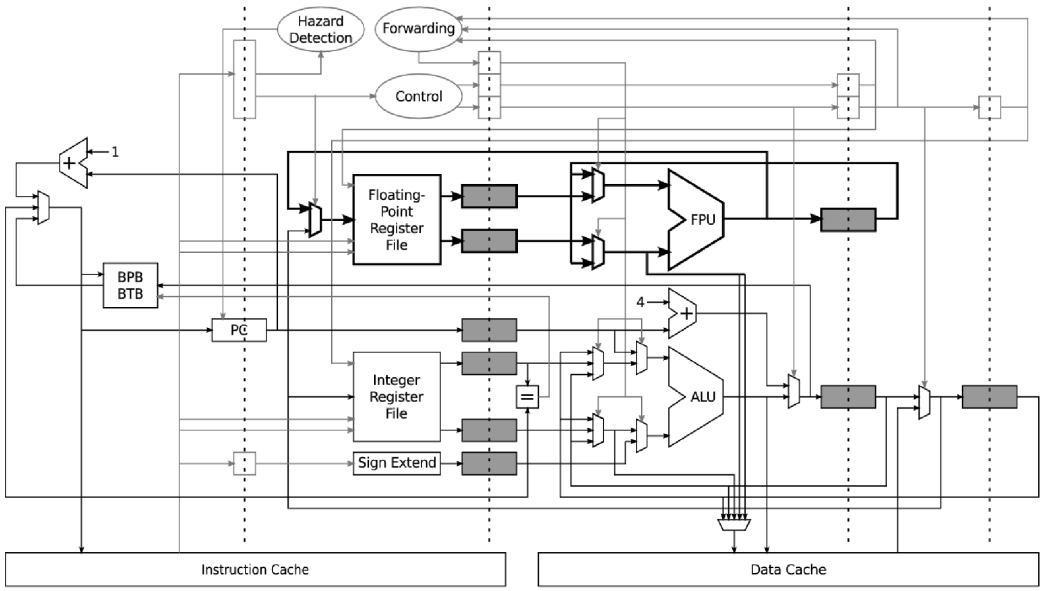
Concepts Introduced in Chapter 10

- instruction pipelining
- multi-issue processors
- detecting data dependences
- eliminating data dependences
- local code scheduling
- global code scheduling
- optimizations to increase ILP

Classical Instruction Pipeline



Scalar Pipeline Diagram

	clock cycle								
	1	2	3	4	5	6	7	8	9
inst 1	IF	RF	EX	MEM	WB				
inst 2		IF	RF	EX	MEM	WB			
inst 3			IF	RF	EX	MEM	WB		
inst 4				IF	RF	EX	MEM	WB	
inst 5					IF	RF	EX	MEM	WB

Superscalar Pipeline Diagram

	clock cycle								
	1	2	3	4	5	6	7	8	9
inst 1	IF	RF	EX	MEM	WB				
inst 2	IF	RF	EX	MEM	WB				
inst 3		IF	RF	EX	MEM	WB			
inst 4		IF	RF	EX	MEM	WB			
inst 5			IF	RF	EX	MEM	WB		
inst 6			IF	RF	EX	MEM	WB		
inst 7				IF	RF	EX	MEM	WB	
inst 8				IF	RF	EX	MEM	WB	
inst 9					IF	RF	EX	MEM	WB
inst 10					IF	RF	EX	MEM	WB

Scheduling Instructions

- Pipeline hazards between instructions can cause stalls.

	clock cycle						
	1	2	3	4	5	6	7
`r[2]=M[r[4]];`	IF	RF	EX	MEM	WB		
`r[3]=r[2]+r[5];`		IF	RF	stall	EX	MEM	WB

- Scheduling instructions is performed by a compiler to:
  - Avoid or reduce the number of stall cycles due to pipeline hazards.
  - Exploit multiple issue capabilities of a processor.
    - superscalar
    - VLIW

Data Dependences between Instructions

- A true dependence occurs when an instruction writes to a location that may be read by a later instruction.

```
r[2] = M[r[3]];
r[2] = r[2] + 1;
```

- An antidependence occurs when an instruction reads from a location that may be written by a later instruction.

```
r[3] = r[2] + 1;
r[2] = r[4] + r[5];
```

- An output dependence occurs when an instruction writes to a location that may be written by a later instruction.

```
M[r[3]] = r[2];
M[r[4]] = r[5];
```

Detecting Dependences between Memory References

- Detecting dependences between memory references is more difficult as the addresses are often unknown until run time.
- Classes of memory access dependency analysis:
  - Simple analysis
    - Different offsets of the same register.

      `M[r[2]+4]` and `M[r[2]+8]`
    - Different base addresses (e.g. stack vs global)

      `M[r[sp]+120] and M[_a+12]`

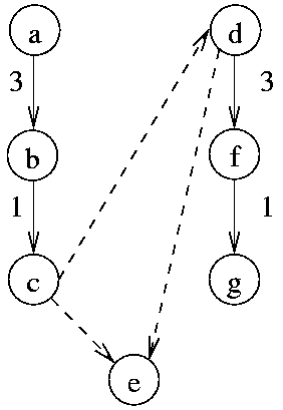
Detecting Dependences between Memory References (cont.)

- array data dependence analysis
  - Detect dependences between two array references in the same loop.
  - Requires solving a set of equations to determine if the references can refer to the same address at the same time.
- points-to analysis
  - Requires detecting the range of addresses to which a pointer can reference.
  - Typically requires interprocedural analysis.

Representing Data Dependences

- The dependences between instructions within a basic block are represented in a DAG with weights on the edges to represent latency of operations.

```
a. r[2]=M[r[4]];
   stall
   stall
b. r[2]=r[2]+r[3];
c. M[r[4]]=r[2];
d. r[2]=M[r[4]+4];
e. r[4]=r[4]+8;
   stall
f. IC=r[2]?0;
g. PC=IC!=0→L13;
```



Eliminating False Data Dependences

- True dependences cannot be eliminated.
- False (anti and output) dependences can be eliminated by renaming the location being referenced.

```
r[2]=M[r[4]];      r[2]=M[r[4]];      r[2]=M[r[4]];
stall              stall              r[5]=M[r[4]+4];
stall              stall              r[4]=r[4]+8;
r[2]=r[2]+r[3];    r[2]=r[2]+r[2];    r[2]=r[2]+r[3];
M[r[4]]=r[2];   ⇒  M[r[4]]=r[2];   ⇒  M[r[4]-8]=r[2];
r[2]=M[r[4]+4];    r[5]=M[r[4]+4];    IC=r[5]?0;
r[4]=r[4]+8;       r[4]=r[4]+8;       PC=IC!=0→L13;
stall              stall
IC=r[2]?0;         IC=r[5]?0;
PC=IC!=0→L13;      PC=IC!=0→L13;
```

Scheduling for a Multi-Issue Processor

- Often a resource reservation table is used when scheduling for a multi-issue processor.
- Assume one cycle load stall and at most one memory reference and one other instruction per cycle.

```
Orig Insts         Mem Insts          Other Insts
r[2]=M[r[4]];      r[2]=M[r[4]];      r[4]=r[4]+8;
stall              r[5]=M[r[4]-4];
r[2]=r[2]+r[3];                       r[2]=r[2]+r[3];
M[r[4]]=r[2];   ⇒  M[r[4]-8]=r[2];    IC=r[5]?0;
r[2]=M[r[4]+4];                       PC=IC!=0,L13;
r[4]=r[4]+8;
IC=r[2]?0;
PC=IC!=0→L13;
```

Global Instruction Scheduling

- Often basic blocks do not have enough ILP for a multi-issue processor to utilize most resources.
- Global scheduling attempts to move instructions from one block into a different block (across basic block boundaries).
- Two blocks are control independent if they are both executed when the other block is executed.
- The scheduler must ensure that an instruction moved from a control-dependent block cannot:
  - change an input value to a different block.
  - cause an exception.

Global Instruction Scheduling Example

- Assume one cycle load stalls and the ability to issue any two instructions together. With only scheduling within each block there are limited opportunities.

```
   PC=IC==0,L5;                 PC=IC==0,L5;
   r[7]=M[r[2]];                r[7]=M[r[2]];
   stall                        stall
   r[6]=r[7]+2;                 r[6]=r[7]+2;      PC=L6;
   PC=L6;
L5:                          L5:
   r[7]=M[r[4]];        ⇒       r[7]=M[r[4]];
   stall                        stall
   r[6]=r[7]+1;                 r[6]=r[7]+1;
L6:                          L6:
   r[3]=r[3]+4;                 r[3]=r[3]+4;
   r[5]=r[3];                   r[5]=r[3];        M[r[3]]=r[4];
   M[r[3]]=r[4];
```

Global Instruction Scheduling Example (cont.)

- We can increase the ILP with global scheduling.

```
   PC=IC==0,L5;                 PC=IC==0,L5;
   r[7]=M[r[2]];                r[7]=M[r[2]];   r[3]=r[3]+4;
   stall                        r[5]=r[3];      M[r[3]]=r[4];
   r[6]=r[7]+2;                 PC=L6;          r[6]=r[7]+2;
   PC=L6;
L5:                          L5:
   r[7]=M[r[4]];        ⇒       r[7]=M[r[4]];   r[3]=r[3]+4;
   stall                        r[5]=r[3];      M[r[3]]=r[4];
   r[6]=r[7]+1;                 r[6]=r[7]+1;
L6:
   r[3]=r[3]+4;                 L6:
   r[5]=r[3];
   M[r[3]]=r[4];
```

Increasing Available ILP for Scheduling

- Often instructions in different loop iterations are independent.
- Loop unrolling merges multiple original iterations and increases basic block size.
- Software pipelining reorganizes loop iterations by peeling a portion of the initial iteration into the loop prologue and a portion of the last iteration into the loop epilogue.

Increasing ILP by Loop Unrolling

- Assume one cycle loads and a two issue processor.

```
   original loop                after scheduling
L2:                          L2:
   r[2]=M[r[4]];                r[2]=M[r[4]];   r[4]=r[4]+4;
   stall                        IC=r[4]?r[5];
   r[3]=r[3]+r[2];              r[3]=r[3]+r[2]; PC=IC!=0,L2;
   r[4]=r[4]+4;
   IC=r[4]?r[5];
   PC=IC!=0,L2;
```

Increasing ILP by Loop Unrolling (cont.)

unrolled loop

```
L2:
   r[2]=M[r[4]];
   stall
   r[3]=r[3]+r[2];
   r[2]=M[r[4]+4];
   stall
   r[3]=r[3]+r[2];
   r[4]=r[4]+8;
   IC=r[4]?r[5];
   PC=IC!=0,L2;
```

after scheduling

```
L2:
   r[2]=M[r[4]];    r[4]=r[4]+8;
   r[6]=M[r[4]-4];  IC=r[4]?r[5];
   r[3]=r[3]+r[2];
   r[3]=r[3]+r[6];  PC=IC!=0,L2;
```

Increasing ILP by Software Piplining

- Assume two cycle loads and a two issue processor.

original loop

```
L2:
   r[2]=M[r[4]];
   stall
   stall
   r[3]=r[3]+r[2];
   r[4]=r[4]+4;
   IC=r[4]?r[5];
   PC=IC!=0,L2;
```

after scheduling

```
L2:
   r[2]=M[r[4]];
   r[4]=r[4]+4;
   IC=r[4]?r[5];
   r[3]=r[3]+r[2];  PC=IC!=0,L2;
```

Increasing ILP by SW Piplining (cont.)

SW pipelined loop

```
   r[2]=M[r[4]];
   r[4]=r[4]+4;
   r[5]=r[5]-4;
```

after scheduling

```
   r[2]=M[r[4]];
   r[4]=r[4]+4;
   r[5]=r[5]-4;
```

```
L2:
   r[3]=r[3]+r[2];
   IC=r[4]?r[5];
   r[2]=M[r[4]];
   r[4]=r[4]+4;
   PC=IC!=0,L2;
```

```
L2:
   r[3]=r[3]+r[2];  r[2]=M[r[4]];
   IC=r[4]?r[5];    r[4]=r[4]+4;
   PC=IC!=0,L2;
```

```
   r[3]=r[3]+r[2];
```

```
   r[3]=r[3]+r[2];
```