CSc 553 — Principles of Compilation

25 : Instruction Scheduling II

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

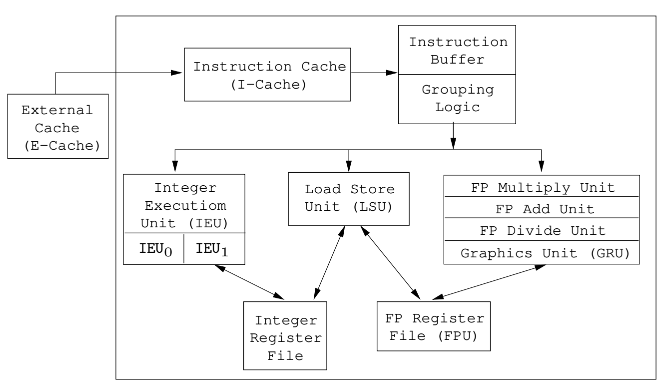
March 24, 2011

1 Introduction

2 The UltraSparc-II*i* I

- The integer unit has 2 ALUs for arithmetic, shift, and logical operations. Each has a 9-stage pipeline.
- The Load/Store unit can issue one load or store per cycle.
- The Floating point unit has five separate functional units. Two FP instructions can be issued per cycle. Most FP instructions have a throughput of 1 cycle, and a latency of 3 cycles.
- There's also a graphics unit that can issue 2 instructions per cycle.
- In total, up to 4 instructions can be issued per cycle.

3 The UltraSparc-II*i* II



4 Superscalar Dispatch

- Instructions are *grouped*, fetched, and issued as a block of max k instructions.

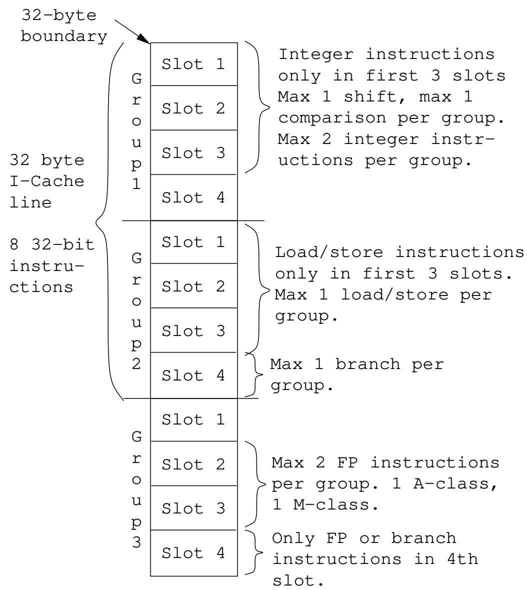
Grp	Cycle	Instruction		Unit
1	1	IntAdd	$\cdots$	ALU_0
	1	IntAdd	$\cdots$	ALU_1
	1	LoadOp	$\cdots$	LSU
	1	FPOp	$\cdots$	FPU_0
2	2	IntAdd	$\cdots$	ALU_0
	2	IntAdd	$\cdots$	ALU_1
	2	LoadOp	$\cdots$	LSU
3	3	LoadOp	$r_1 \leftarrow \cdots$	LSU
4	5	IntAdd	$r_2 \leftarrow \cdots r_1$	ALU_0
	5	IntMul	$\cdots$	MUL
5	41	IntAdd	$\cdots$	ALU_1

5 The UltraSparc-II*i* III

Group	Instruction		Unit
Grp1	add	%o1,55,%o2	ECU_0
	shl	%g2,2,%g2	ECU_1
	ld	[%fp+8],%l2	LSU
	fadd	%f2,%f2,%f4	FP-Add
Grp2	sub	%l1,5,%l1	ECU_0
	add	%l1,20,%o2	ECU_1
	ld	[%fp+8],%l2	LSU

- The first 3 slots of a group can hold most types of instructions, except there can be only one ECU_0 and one ECU_1 instruction per group.
- The fourth slot can only hold a branch or an FP instruction.

6 The UltraSparc-II*i* IV



7 The UltraSparc-II*i* `IEU` I

- Max 2 integer instructions can be issued per cycle. They're dispatched only if they're in the first 3 instruction slots of the group.
- There are two integer pipelines, $\texttt{IEU}_0$ and $\texttt{IEU}_1$.
- Some instructions can only go to one pipeline. `ADD`, `AND`, `ANDN`, `OR`, `ORN`, `SUB`, `XOR`, `XNOR`, `SETHI` can go to either.
- $\texttt{IEU}_0$ has special hardware for shift instructions. Two shift instructions can't be grouped together.

8 The UltraSparc-II*i* `IEU` II

- $\texttt{IEU}_1$ has special hardware for instructions that set condition codes: `ADDcc`, `ANDcc`, `ANDNcc`, `ORcc`, `ORNcc`, `SUBcc`, `XORcc`, `XNORcc`. `CALL`, `JUMPL`, `FCMP` also use the $\texttt{IEU}_1$.
- Two instructions that use the $\texttt{IEU}_1$ can't be grouped together. For example, only one instruction that sets condition codes can be issued per cycle.
- Some instructions execute for several cycles: `MULScc` inserts 1 bubble after it's dispatched. `SDIV` inserts 36 bubbles, `UDIV` insterts 37 bubbles, `DIVX` insterts 68 bubbles.

9 The UltraSparc-II*i* `IEU` III

- Some instructions must complete before another instruction can be dispatched: Depending on the value of the multiplicand, `SMUL` inserts max 18 bubbles, `UMUL` 19 bubbles, `MULX` 34 bubbles.
- Some instructions are *single group*, they're always issued by themselves: `LDD`, `STD`, `ADDC`, `SUBC`, , `MOVcc`, `FMOVcc`, `MOVr`, `SAVE`, `RESTORE`, `UMUL`, `SMUL`, `MULX`, `UDIV`, `SDIV`, `UDIVX`, `SDIVX`.

10 The UltraSparc-II*i* IEU IV

- IEU instructions that write to the same register can't be grouped together:

Group	Instruction		Unit
Grp1	add	%o1,55,%i6	ECU_0
Grp2	ldx	[%l6+0],%i6	LSU

- If IEU instruction (a) reads a register that instruction (b) writes, they can't be grouped together:

Group	Instruction		Unit
Grp1	add	%o1,55,%i6	ECU_0
Grp2	ldx	[%i6+0],%o3	LSU

- In other words, there's a one cycle delay between an instruction that computes a value and an instruction that uses that value.

11 The UltraSparc-II*i* CTI I

- At most one *control transfer instruction* (CTI) can be dispatched per group: CALL, BPcc, Bicc, FB(P)cc, BPr, JMPL.
- BPcc are the *branch on integer condition codes with prediction* instructions: BPA, BPG, BPGE, $\cdots$.
- If the branch is predicted taken, the branch instruction and the instruction at the branch target can be in the same group:

Group	Instruction	Unit
Grp1	*setcc*	ECU_1
	BPcc	CTI
	FADD	FPU (delay slot)
	FMUL	FPU (branch target)

12 The UltraSparc-II*i* CTI II

- If the branch is predicted *not* taken, the branch instruction and the following instruction can be in the same group:

Group	Instruction	Unit
Grp1	*setcc*	ECU_1
	BPcc	CTI
	FADD	FPU (delay slot)
	FMUL	FPU (sequential)

13 The UltraSparc-II*i* LSU I

- Load/store instructions can only be dispatched if they're in the first three instruction slots of a group.
- There can be one load/store dispatched per group.
- An instruction that references the result of a load cannot be in the load-group or the next group:

Group	Instruction		Unit
Grp1	LDDF	[r1],f6	LSU
Grp2			
Grp3	FMULD	f4,f6,f8	FPU

In other words, there's a two cycle load-delay.

14 The UltraSparc-II*i* FPU I

- FP instructions fall in two classes, A and M. An A and an M instruction can be in the same group, but not two A or two M instructions.
- The A class: `FxTOy`, `FABS`, `FADD`, `FAND`, `FCMP`, `FMOV`, `FNEG`, `FSUB`.
- The M class: `FCMP`, `FDIST`, `FDIV`, `FMUL`, `FSQRT`.
- `FPU` instructions that write to the same register can't be grouped together:

Group	Instruction		Unit
Grp1	FADDD	f2,f2,f6	FPU
Grp2	LDF	[%l6+0],f6	LSU

15 The UltraSparc-II*i* FPU I

- A FP store can be in the same group as the instruction that computes the value:

Group	Instruction		Unit
Grp1	FADDD	f2,f2,f6	FPU
	STD	f6,[%l6+0]	LSU

- Most FP instructions have a latency of 3 cycles. I.e., the result generated by instruction (a) cannot be referenced by instruction (b) until 3 cycles later:

Group	Instruction		Unit
Grp1	FADDD	f2,f4,f6	FPU
Grp2			
Grp3			
Grp4	FMULD	f4,f6,f8	FPU

- `FDIVD` and `FSQRTD` have 22-cycle latencies.

16 Instruction Scheduling

The purpose of instruction scheduling is

1. to avoid pipeline stalls due to a datum not being available when needed, and
2. to keep all functional units busy, i.e. making sure that every functional unit has at least one instruction ready to execute, and
3. to fill branch delay slots.

- We'll consider an algorithm, *List Scheduling*, that produces a toplogical sort of the dependence graph, while minimizing the execution time of the basic block.

17

Dependence Graph

18 Building the Dependence Graph I

- There's an edge $a \rightarrow b$ between instructions a and b of the dependence graph if

1. a writes to a register or location that b uses:

$$\begin{array}{ll} (a) & r_1 \leftarrow \cdots \\ (b) & \cdots \leftarrow r_1 \end{array}$$

2. a uses a register or location that b writes to:

$$\begin{array}{ll} (a) & \cdots \leftarrow [r_1 + 16] \\ (b) & [r_1 + 16] \leftarrow \cdots \end{array}$$

3. a and b write to the same register or location:

$$\begin{array}{ll} (a) & r_1 \leftarrow \cdots \\ (b) & r_1 \leftarrow \cdots \end{array}$$

4. we don't know if a can be moved after b:

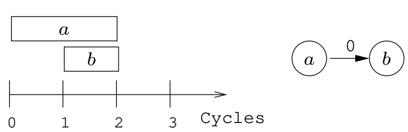
$$\begin{array}{ll} (a) & [r_1 + 16] \leftarrow \cdots \\ (b) & \cdots \leftarrow [r_2 + 32] \end{array}$$

19 Building the Dependence Graph II

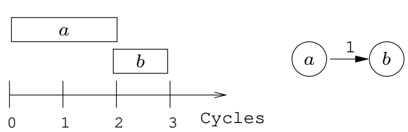
- The edge $a \rightarrow b$ is labeled with an integer *latency*,

  the *delay* required between the initiation times of a and b, minus the execution time required by a before any other instruction can begin executing.

- If b can begin executing in the cycle after a began executing, the latency is 0:



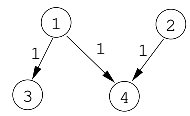
- If two cycles have to elapse between starting a and starting b, the latency is 1:



20 Dependence Graph Example

- Assume that a load has a latency of one cycle and takes two cycles to complete.

```
(1)      load  r2, [r1+4]
(2)      load  r3, [r1]
(3)      add   r4, r2, r3
(4)      sub   r5, r2, 1
```



Note:

- When building the graph we must take implicit resources like condition codes into account:

  5. There's an edge $a \rightarrow b$ if a sets a condition code and b brances on it.

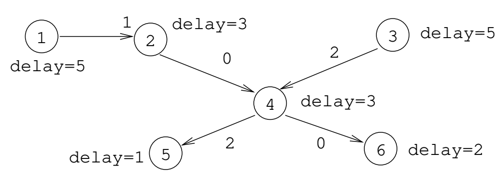
21

List Scheduling Algorithm

22 List Scheduling Algorithm I

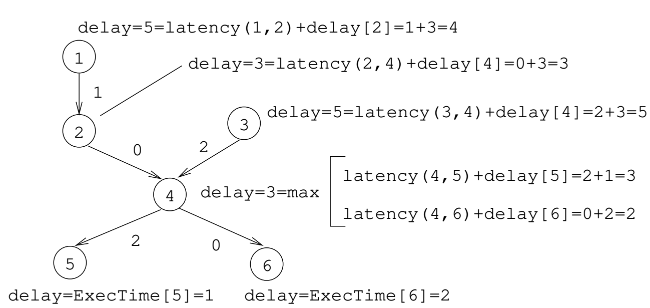
- Consider the dependence graph below. Let `ExecTime[6]=2`, and `ExecTime=1` for all other instructions.
- Start by labeling the nodes with the maximum possible *delay* from the node to the end of the block.

$$\texttt{delay}[n] = \begin{cases} \texttt{ExecTime}[n] \text{if } n & \text{is a leaf} \\ \max\limits_{\text{succs } m \text{ of } n} & (\texttt{latency}(n,m) + \texttt{delay}[m]) \quad \text{otherw.} \end{cases}$$



23 Example I

- Consider the dependence graph below. Let `ExecTime[6]=2`, and `ExecTime=1` for all other instructions.



24 List Scheduling Algorithm II

- Next, traverse the graph from the root to the leaves.
- Select nodes to schedule.
- Keep track of the current time, `CurTime`.
- `ETime[`n`]` is the earliest time node n should be scheduled to avoid a stall.
- `Candidates` is the set of candidates (nodes which can be scheduled).
- `MCands` is the set of candidates with the maximum delay time to the end of the block.
- `ECands` is the set of candidates whose earliest start times are $\leq$ `CurTime`.

25

```
REPEAT
   Candidates := nodes in DAG with indegree=0;
   MCands := Candidates with max Delay;
   ECands := Candidates whose ETime ≤ CurTime;
   IF there's just one m ∈ MCands THEN n := m
   ELSIF there's just one m ∈ ECands THEN n := m
   ELSIF there's more than one m ∈ MCands THEN
      n := Heuristics(MCands)
   ELSIF there's more than one m ∈ ECands THEN
      n := Heuristics(ECands) ENDIF;
   Schedule n;
   CurTime := CurTime + ExecTime[n];
   FOR every successor i of n DO
      ETime[i] := max(ETime[n],CurTime+Latency(n,i))
```

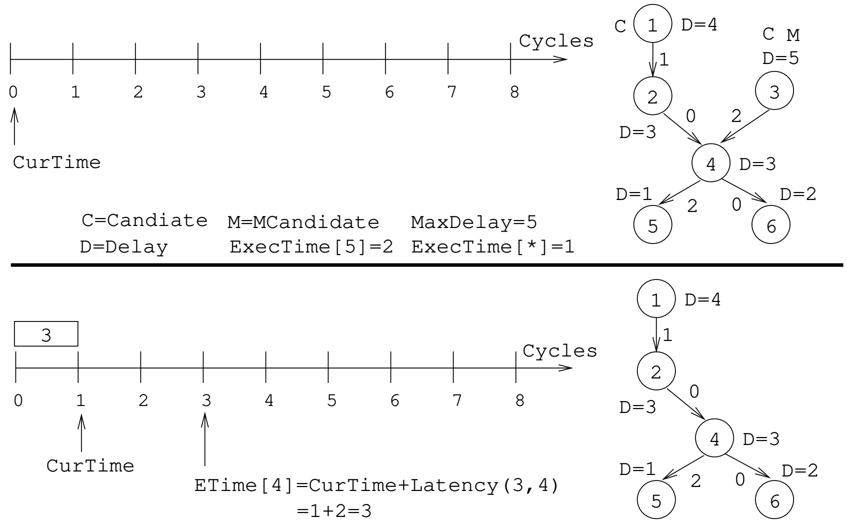
26 List Scheduling Algorithm IV

- As usual, there are many possible heuristics to choose from:

`PROCEDURE Heuristics (`M` : SET OF Nodes) : Node`

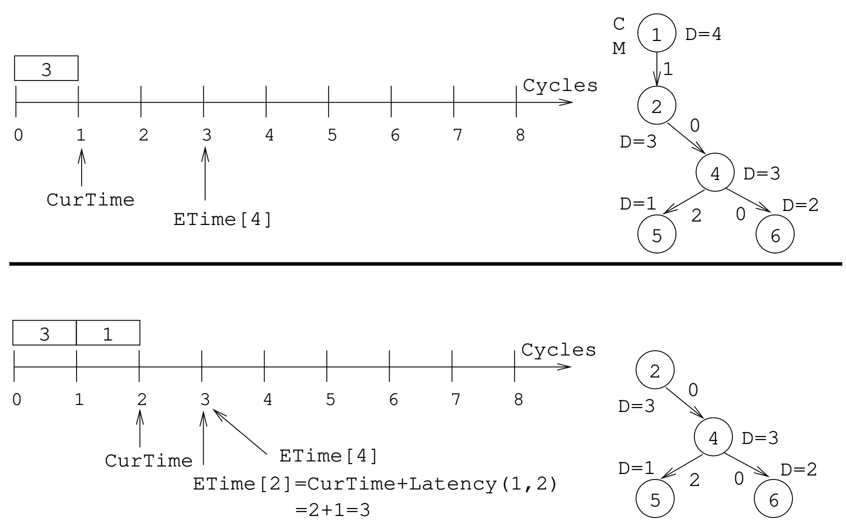
- Pick the n from `MCands` with minimum `ETime`[n].
- Pick the n with maximum total delay to the leaves.
- Pick the n that adds the most new candidates.
- Pick the n that originally came first in the basic block.

27



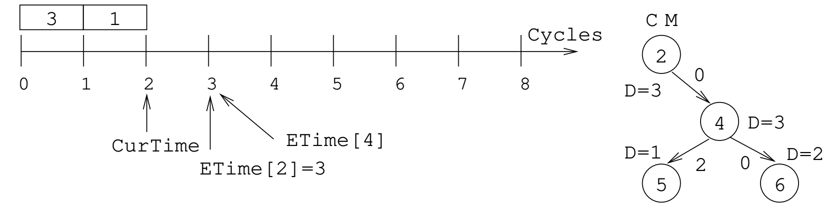
Cycles
0 1 2 3 4 5 6 7 8
CurTime
C=Candiate D=Delay
M=MCandidate ExecTime[5]=2
MaxDelay=5 ExecTime[*]=1
C 1 D=4
C M D=5 3
2 D=3
4 D=3
5 D=1
6 D=2
3
ETime[4]=CurTime+Latency(3,4)
=1+2=3

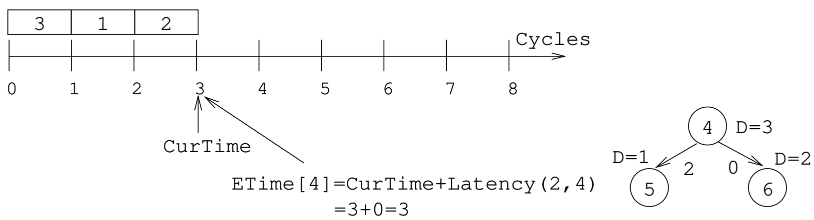
28



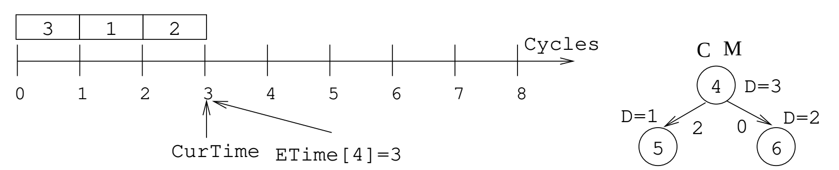
3
Cycles
0 1 2 3 4 5 6 7 8
CurTime
ETime[4]
C M 1 D=4
2 D=3
4 D=3
5 D=1
6 D=2
3 1
ETime[4]
ETime[2]=CurTime+Latency(1,2)
=2+1=3

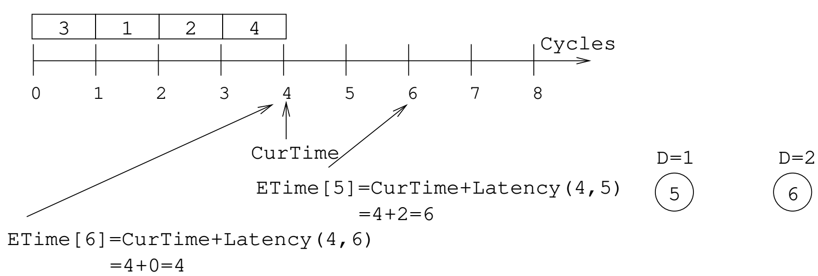
29


3 1
Cycles
0 1 2 3 4 5 6 7 8
CurTime
ETime[4]
ETime[2]=3
C M
D=3
D=1
D=2

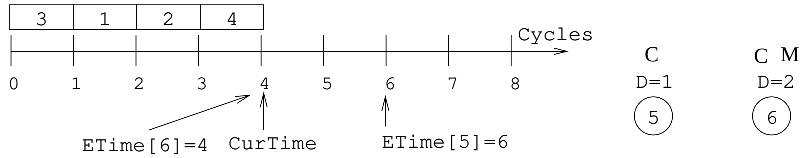

3 1 2
Cycles
0 1 2 3 4 5 6 7 8
CurTime
ETime[4]=CurTime+Latency(2,4)
=3+0=3
D=3
D=1
D=2

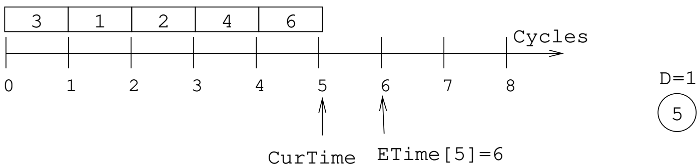
30


3 1 2
Cycles
0 1 2 3 4 5 6 7 8
CurTime
ETime[4]=3
C M
D=3
D=1
D=2

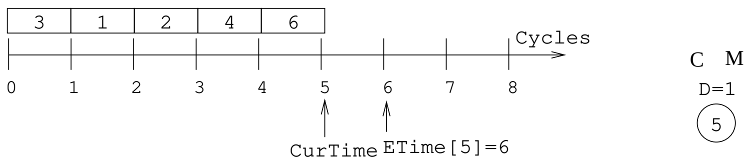

3 1 2 4
Cycles
0 1 2 3 4 5 6 7 8
CurTime
ETime[5]=CurTime+Latency(4,5)
=4+2=6
ETime[6]=CurTime+Latency(4,6)
=4+0=4
D=1
D=2

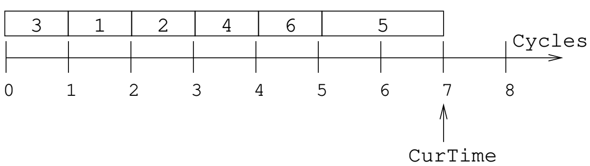
31





32





33 List Scheduling Algorithm V

- How do we deal with superscalar architectures?
- We can easily modify the heuristic to handle $p > 1$ pipelines:

`PROCEDURE Heuristics (`M` : SET OF Nodes) : Node`

- Pick a node n from M such that
  1. instruction n can execute on pipeline P_i, and
  2. pipeline P_i hasn't had an instruction scheduled for it recently.

34 Filling Delay Slots I

- Typically, a basic block ends with a branch:

```
(1)       Instr_1
          ...
(n-1)     Branch
(n)       Delay Slot
```

- We pick an instruction (i) from the basic block to fill the delay slot, such that
  1. (i) is a leaf of the dependence graph (otherwise, some other instruction b depends on it and would have to be executed after the branch), and
  2. the branch must not depend on (i) (e.g. (i) can't set the condition codes that are branched on), and
  3. (i) is not a branch itself.

35 Filling Delay Slots II

- If that doesn't work, we can try to move an instruction from the *target* basic block into the delay slot:

```
(1)       Instr_1
          ...
(n-1)     Branch L
(n)       Delay Slot
          ...
L:        Instr_k  ⇐ move to delay slot!
```

- We prefer a single-cycle instruction (like an integer `add`) over a multi-cycle (like a delayed load).
- If we can't find a suitable instruction, insert a `nop`.

36 Filling Delay Slots III

- Calls are similar:

```
(1)       ArgReg_1 ← ···
(2)       ArgReg_2 ← ···
          ...
(n-1)     Call P
(n)       Delay Slot
```

- There's usually an instruction (i) that moves an argument into one of the argument-passing registers. We prefer to put that instruction in the delay slot.
- If not, we can try to move an instruction *from the next basic block* (the one following the call) into the slot:

```
(n-1)     Call P
(n)       Delay Slot
(n+1)     Instr_k  ⇐ move to delay slot!
```

- Failing that, we insert a `nop`.

37 Readings and References

- Steven Muchnick, *Advanced Compiler Design and Implementation*, Section 9.2, pp. 269–274 and Chapter 17, pp. 531–547.
- Sun Technical manuals for UltraSparc-II*i* (available from `www.sun.com`):
  1. Chapter 1: UltraSparc-II*i* Basics
  2. Chapter 2: Processor Pipeline
  3. Chapter 21: Code Generation Guidelines
  4. Chapter 22: Grouping Rules and Stalls.