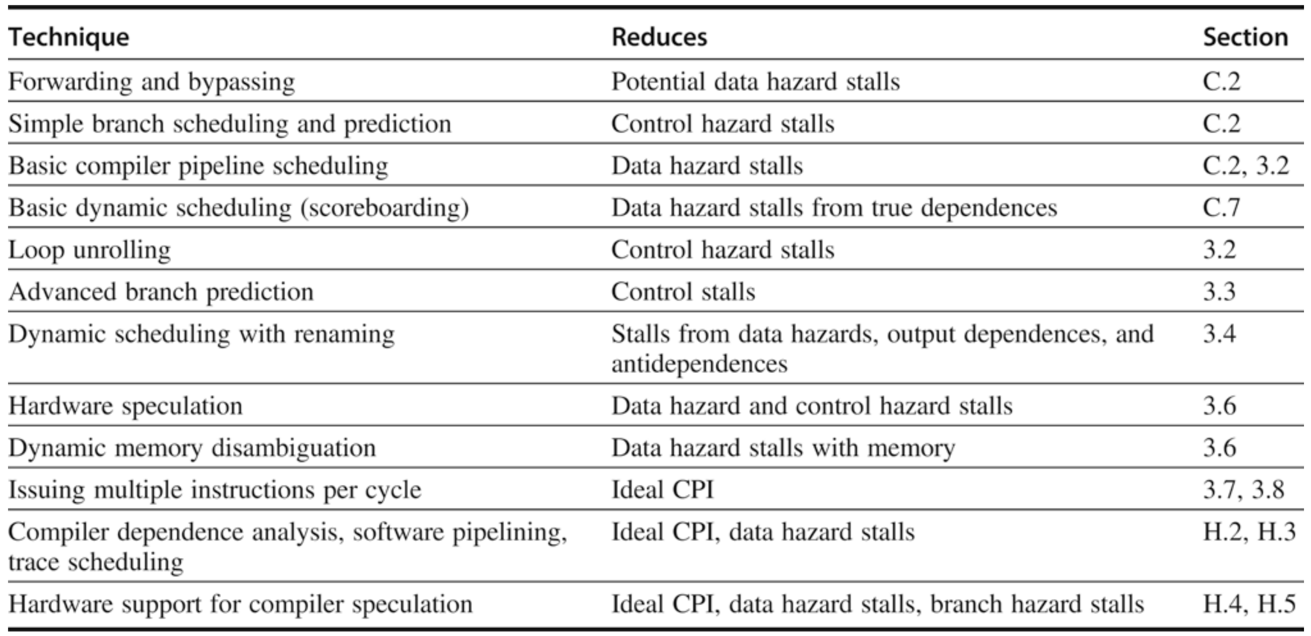
Chapter 3
Instruction-Level Parallelism and Its Exploitation



Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

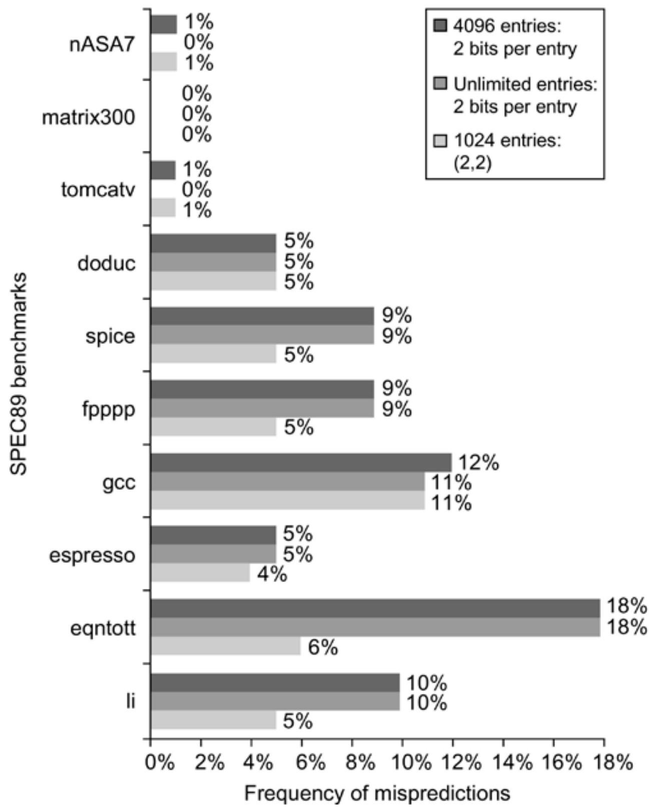


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

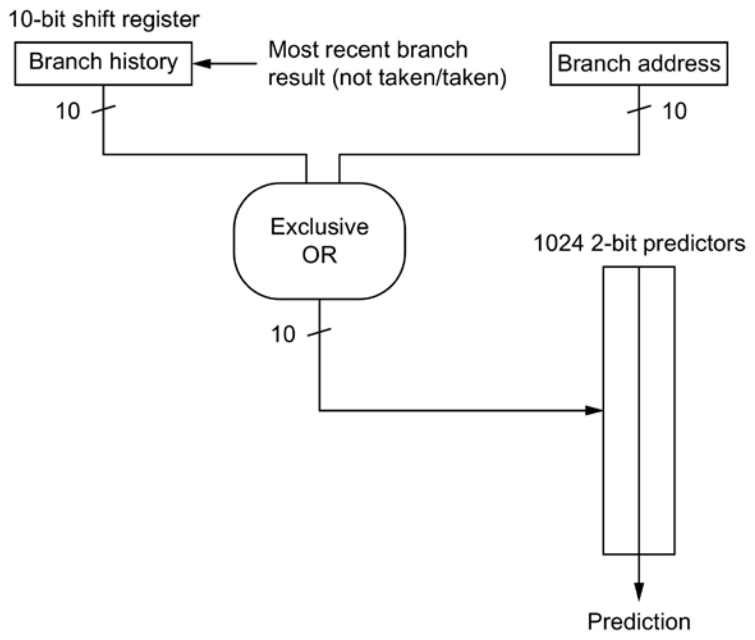


Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

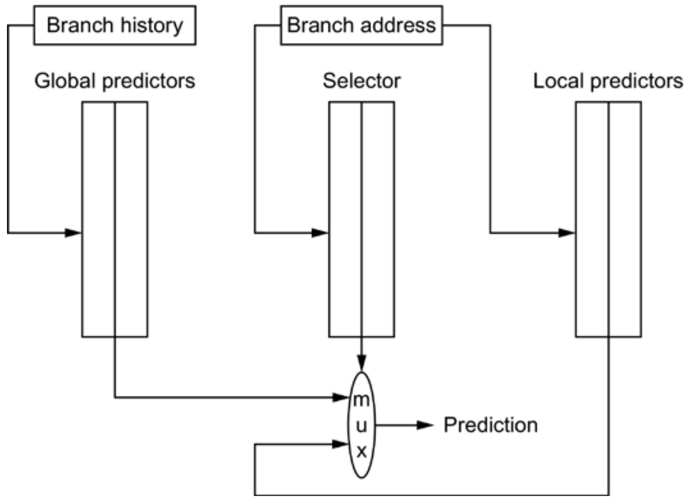


Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

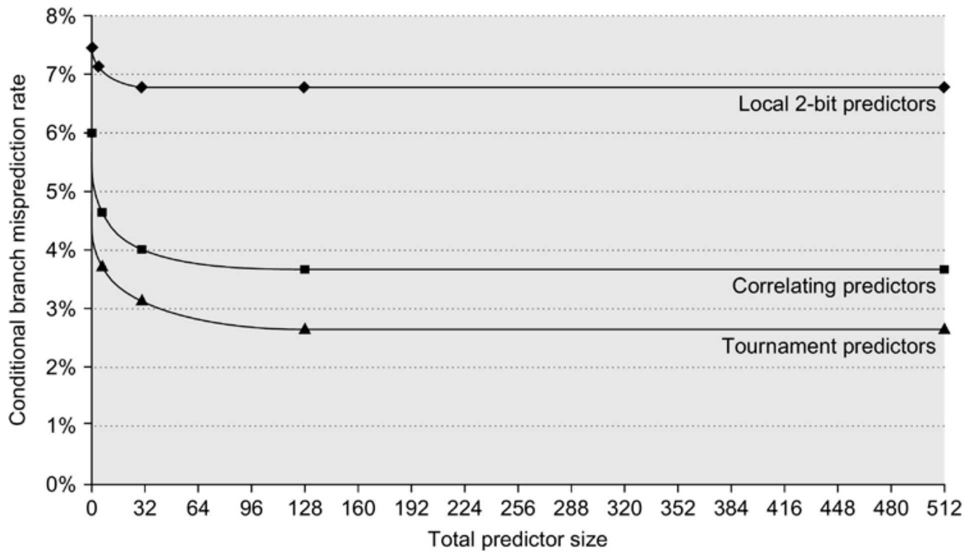


Figure 3.6 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

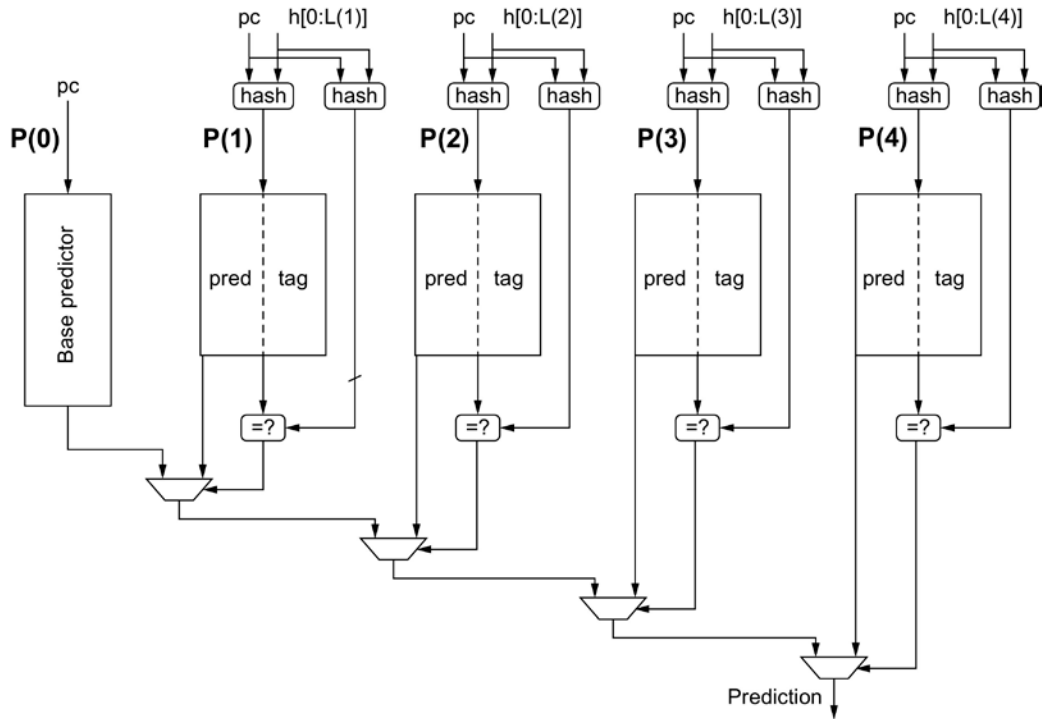


Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

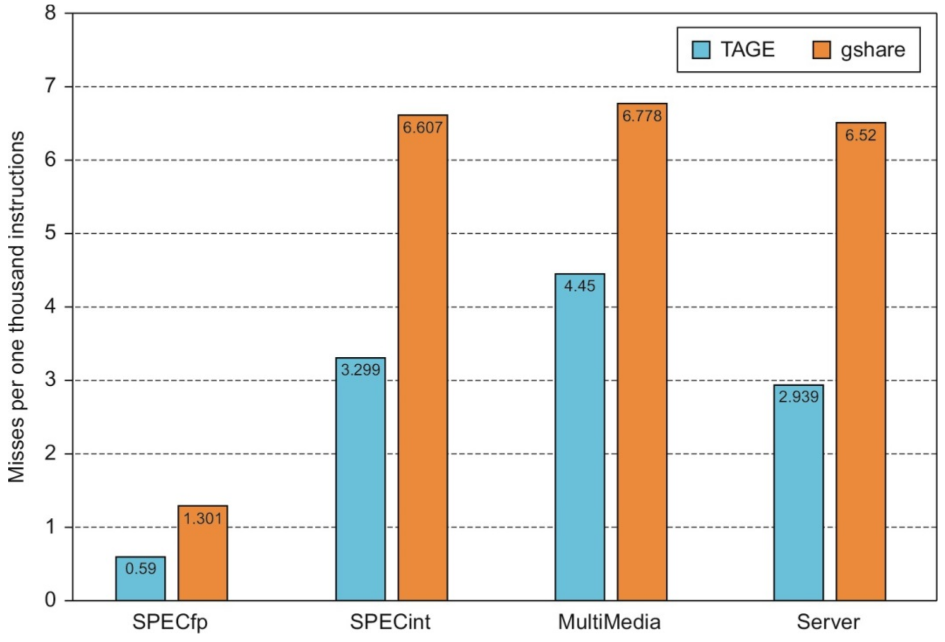


Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

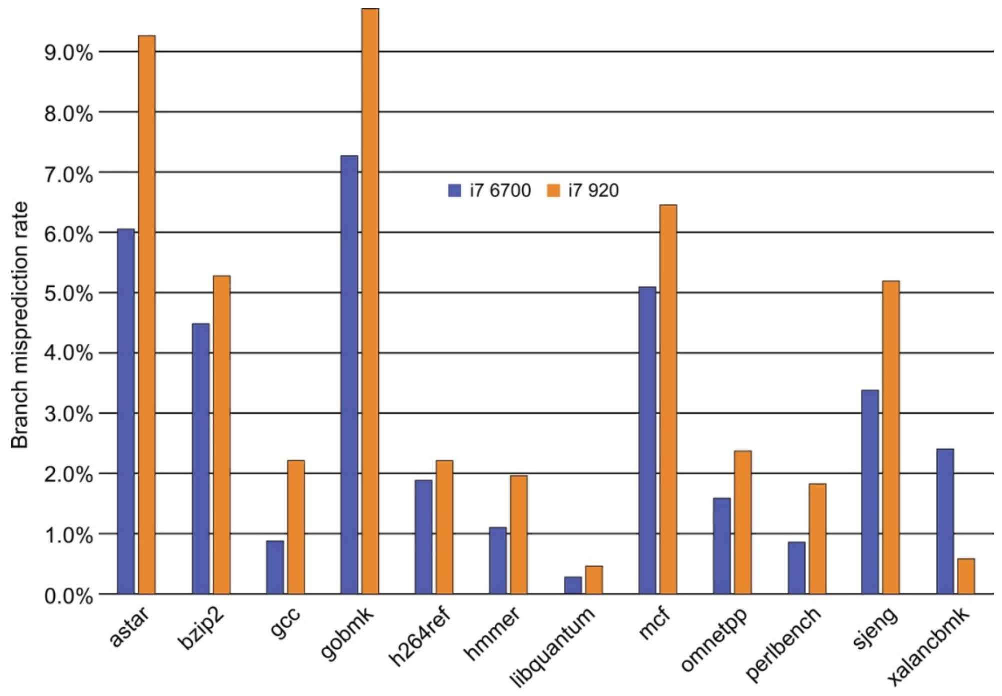


Figure 3.9 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 920 and 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. This could understate the misprediction rate somewhat because if a branch is mispredicted and led to another mispredicted branch (which should not have been executed), it will be counted as only one misprediction. On average, the i7 920 mispredicts branches 1.3 times as often as the i7 6700.

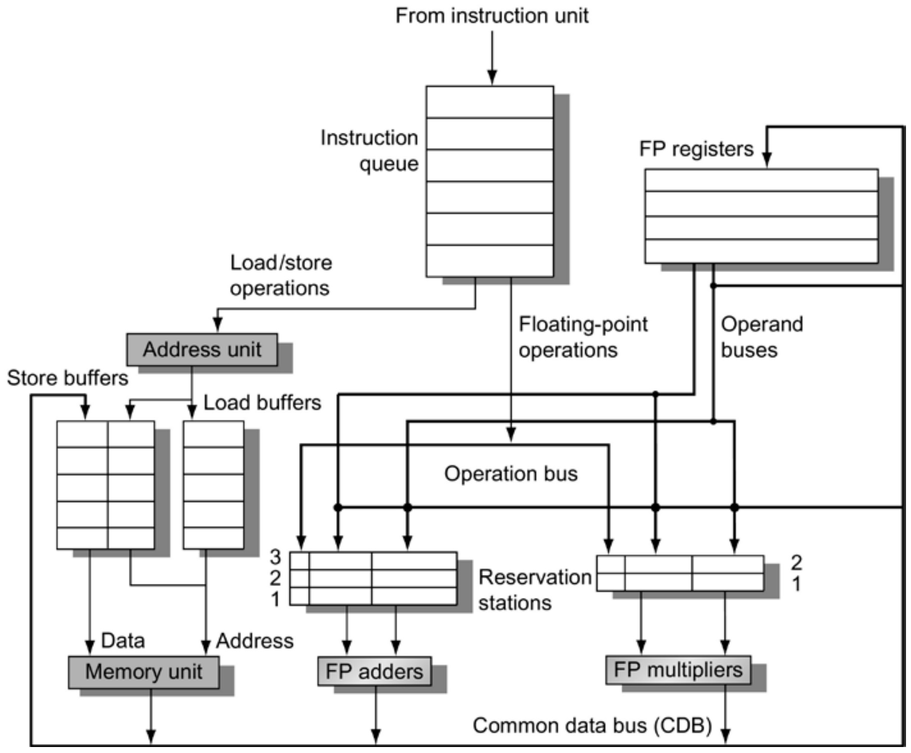


Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: (1) hold the components of the effective address until it is computed, (2) track outstanding loads that are waiting on the memory, and (3) hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: (1) hold the components of the effective address until it is computed, (2) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (3) hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

Instruction		Instruction status		
		Issue	Execute	Write result
`fld`	`f6,32(x2)`	√	√	√
`fld`	`f2,44(x3)`	√	√	
`fmul.d`	`f0,f2,f4`	√		
`fsub.d`	`f8,f2,f6`	√		
`fdiv.d`	`f0,f0,f6`	√		
`fadd.d`	`f6,f8,f2`	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		`Regs[f4]`	Load2		
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

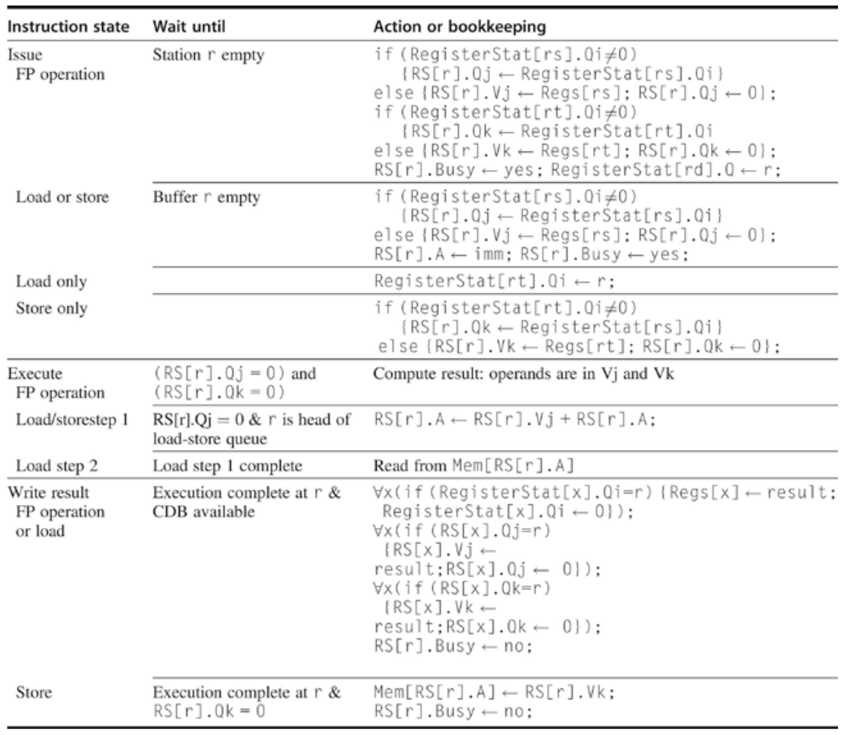
Figure 3.11 Reservation stations and register tags shown when all of the instructions have issued but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the `fadd.d` instruction, which has a WAR hazard at the WB stage, has issued and could complete before the `fdiv.d` initiates.

	Instruction status		
Instruction	**Issue**	**Execute**	**Write result**
`fld f6,32(x2)`	√	√	√
`fld f2,44(x3)`	√	√	√
`fmul.d f0,f2,f4`	√	√	
`fsub.d f8,f2,f6`	√	√	√
`fdiv.d f0,f0,f6`	√		
`fadd.d f6,f8,f2`	√	√	√

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	`Mem[44 + Regs[x3]]`	`Regs[f4]`			
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status								
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	**...**	**f30**
Qi	Mult1					Mult2			

Figure 3.12 Multiply and divide are the only instructions not finished.



Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	`if (RegisterStat[rs].Qi≠0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`if (RegisterStat[rt].Qi≠0)`
		`{RS[r].Qk ← RegisterStat[rt].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`

Load or store	Buffer r empty	`if (RegisterStat[rs].Qi≠0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`RS[r].A ← imm; RS[r].Busy ← yes;`

Load only		`RegisterStat[rt].Qi ← r;`

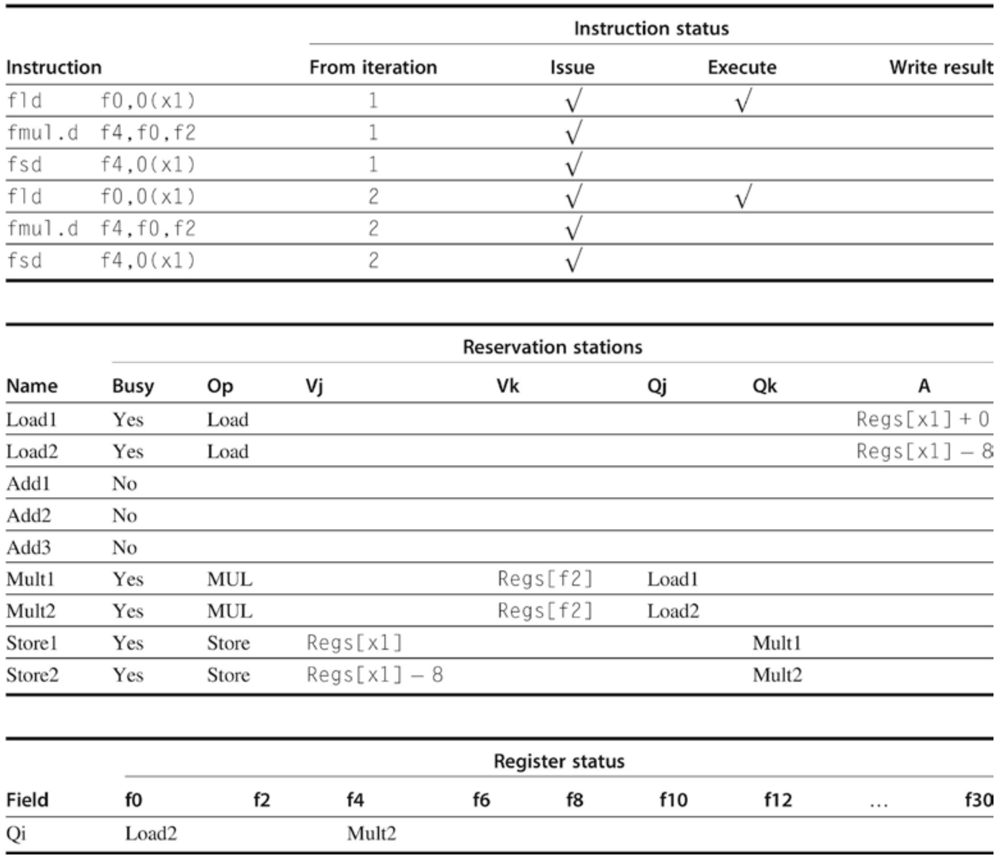
Store only		`if (RegisterStat[rt].Qi≠0)`
		`{RS[r].Qk ← RegisterStat[rs].Qi}`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`

Execute FP operation	`(RS[r].Qj = 0)` and `(RS[r].Qk = 0)`	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`

Write result
FP operation
or load
	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result;`
		`RegisterStat[x].Qi ← 0});`
		`∀x(if (RS[x].Qj=r)`
		`{RS[x].Vj ←`
		`result;RS[x].Qj ← 0});`
		`∀x(if (RS[x].Qk=r)`
		`{RS[x].Vk ←`
		`result;RS[x].Qk ← 0});`
		`RS[r].Busy ← no;`

Store	Execution complete at r & `RS[r].Qk = 0`	`Mem[RS[r].A] ← RS[r].Vk;`
		`RS[r].Busy ← no;`

Figure 3.13 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the source register numbers, `imm` is the sign-extended immediate field, and `r` is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure. The value returned by an FP unit or by the load unit is called `result`. `RegisterStat` is the register status data structure (not the register file, which `is Regs[]`). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose values of Qj or Qk are the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that, to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because no concept of program order is maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step if there is a pending branch already in the pipeline. In Section 3.6, we will see how speculation support removes this restriction.



		Instruction status		
Instruction	From iteration	Issue	Execute	Write result
`fld f0,0(x1)`	1	√	√	
`fmul.d f4,f0,f2`	1	√		
`fsd f4,0(x1)`	1	√		
`fld f0,0(x1)`	2	√	√	
`fmul.d f4,f0,f2`	2	√		
`fsd f4,0(x1)`	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					`Regs[x1] + 0`
Load2	Yes	Load					`Regs[x1] − 8`
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		`Regs[f2]`	Load1		
Mult2	Yes	MUL		`Regs[f2]`	Load2		
Store1	Yes	Store	`Regs[x1]`			Mult1	
Store2	Yes	Store	`Regs[x1] − 8`			Mult2	

	Register status							
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

Figure 3.14 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

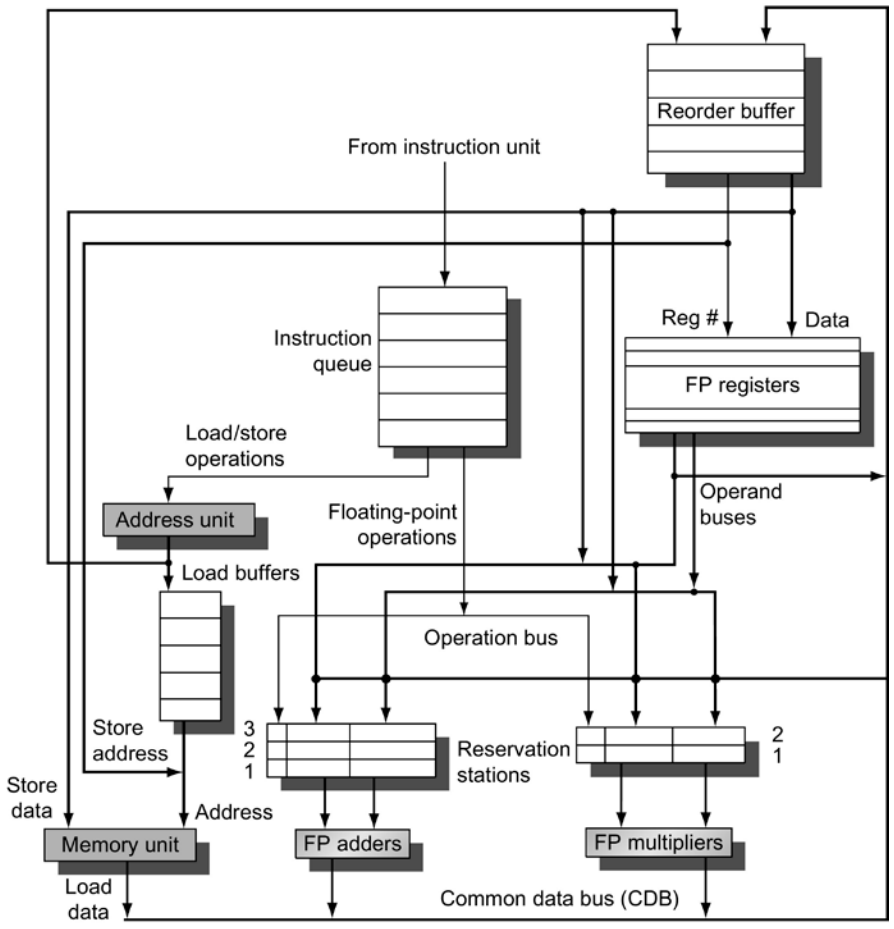
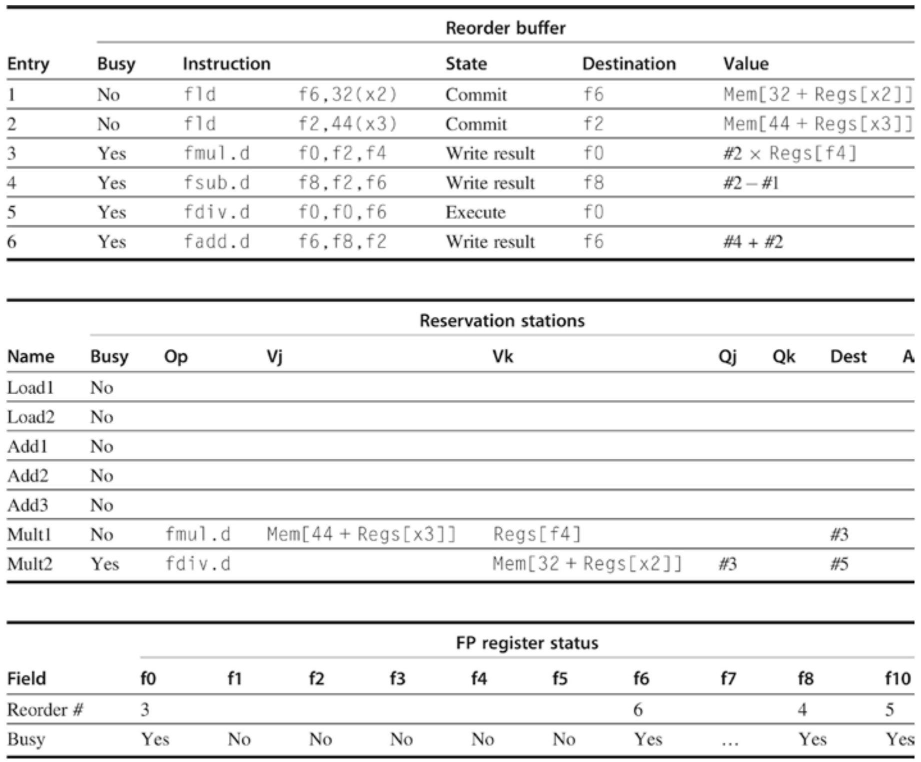


Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.10 on page 198, which implemented Tomasulo's algorithm, we can see that the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to allow multiple issues per clock by making the CDB wider to allow for multiple completions per clock.



	Reorder buffer					
Entry	Busy	Instruction		State	Destination	Value
1	No	`fld`	`f6,32(x2)`	Commit	`f6`	`Mem[32 + Regs[x2]]`
2	No	`fld`	`f2,44(x3)`	Commit	`f2`	`Mem[44 + Regs[x3]]`
3	Yes	`fmul.d`	`f0,f2,f4`	Write result	`f0`	#2 × `Regs[f4]`
4	Yes	`fsub.d`	`f8,f2,f6`	Write result	`f8`	#2 – #1
5	Yes	`fdiv.d`	`f0,f0,f6`	Execute	`f0`	
6	Yes	`fadd.d`	`f6,f8,f2`	Write result	`f6`	#4 + #2

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	`fmul.d`	`Mem[44 + Regs[x3]]`	`Regs[f4]`			#3	
Mult2	Yes	`fdiv.d`		`Mem[32 + Regs[x2]]`	#3		#5	

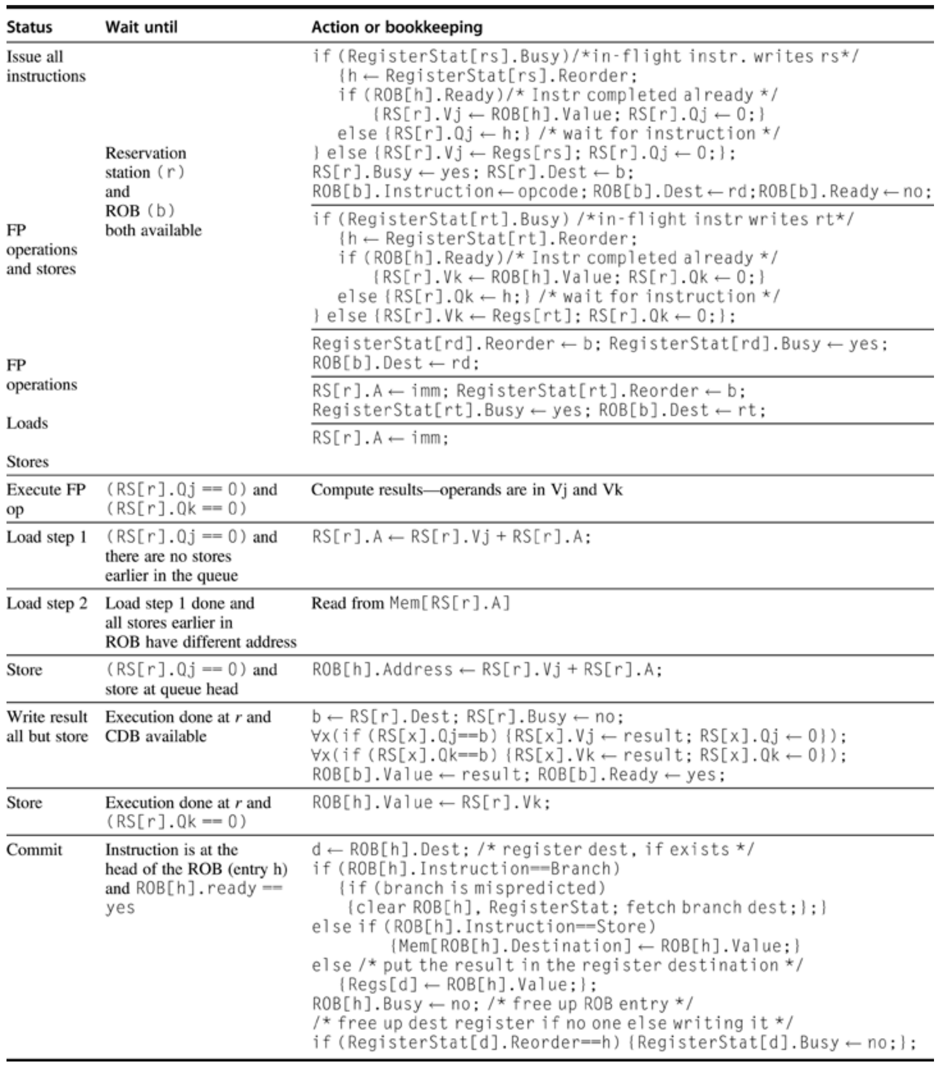
	FP register status								
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

Figure 3.16 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are there only to ease understanding. The `fsub.d` and `fadd.d` instructions will not commit until the `fmul.d` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `fdiv.d` is in execution, but has not completed solely because of its longer latency than that of `fmul.d`. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed but are shown for informational purposes. We do not show the entries for the load/store queue, but these entries are kept in order.

	Reorder buffer				
Entry	**Busy**	**Instruction**	**State**	**Destination**	**Value**
1	No	`fld f0,0(x1)`	Commit	`f0`	`Mem[0 + Regs[x1]]`
2	No	`fmul.d f4,f0,f2`	Commit	`f4`	#1 × `Regs[f2]`
3	Yes	`fsd f4,0(x1)`	Write result	`0 + Regs[x1]`	#2
4	Yes	`addi x1,x1,−8`	Write result	`x1`	`Regs[x1] − 8`
5	Yes	`bne x1,x2,Loop`	Write result		
6	Yes	`fld f0,0(x1)`	Write result	`f0`	`Mem[#4]`
7	Yes	`fmul.d f4,f0,f2`	Write result	`f4`	#6 × `Regs[f2]`
8	Yes	`fsd f4,0(x1)`	Write result	0 + #4	#7
9	Yes	`addi x1,x1,−8`	Write result	`x1`	#4 – 8
10	Yes	`bne x1,x2,Loop`	Write result		

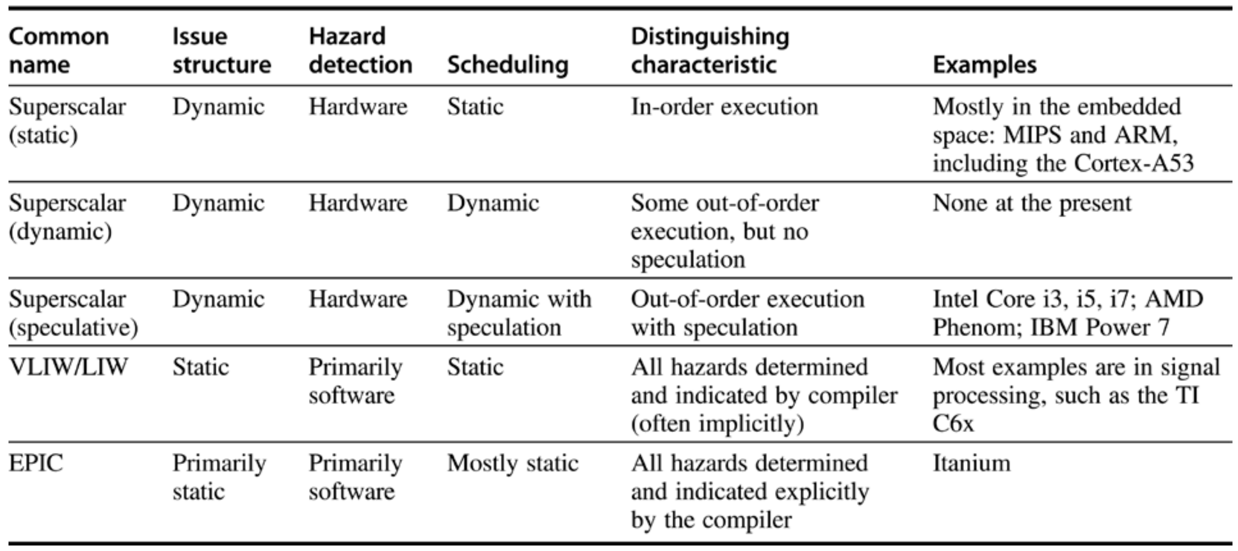
	FP register status								
Field	**f0**	**f1**	**f2**	**f3**	**f4**	**F5**	**f6**	**F7**	**f8**
Reorder #	6								
Busy	Yes	No	No	No	Yes	No	No	…	No

Figure 3.17 Only the `fld` and `fmul.d` instructions have committed, although all the others have completed execution. Thus no reservation stations are busy and none are shown. The remaining instructions will be committed as quickly as possible. The first two reorder buffers are empty, but are shown for completeness.



Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from `Mem[RS[r].A]`
Store	(RS[r].Qj == 0) and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`
Write result all but store	Execution done at *r* and CDB available	`b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;`
Store	Execution done at *r* and (RS[r].Qk == 0)	`ROB[h].Value ← RS[r].Vk;`
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	`d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`

Figure 3.18 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, `r` is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the `result`. `Register-Stat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.



Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

Figure 3.20 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming correct branch prediction. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than RISC-V would normally use in this loop. The preceding VLIW code sequence requires at least eight FP registers, whereas the same code sequence for the base RISC-V processor can use as few as two FP registers or as many as five when unrolled and scheduled.

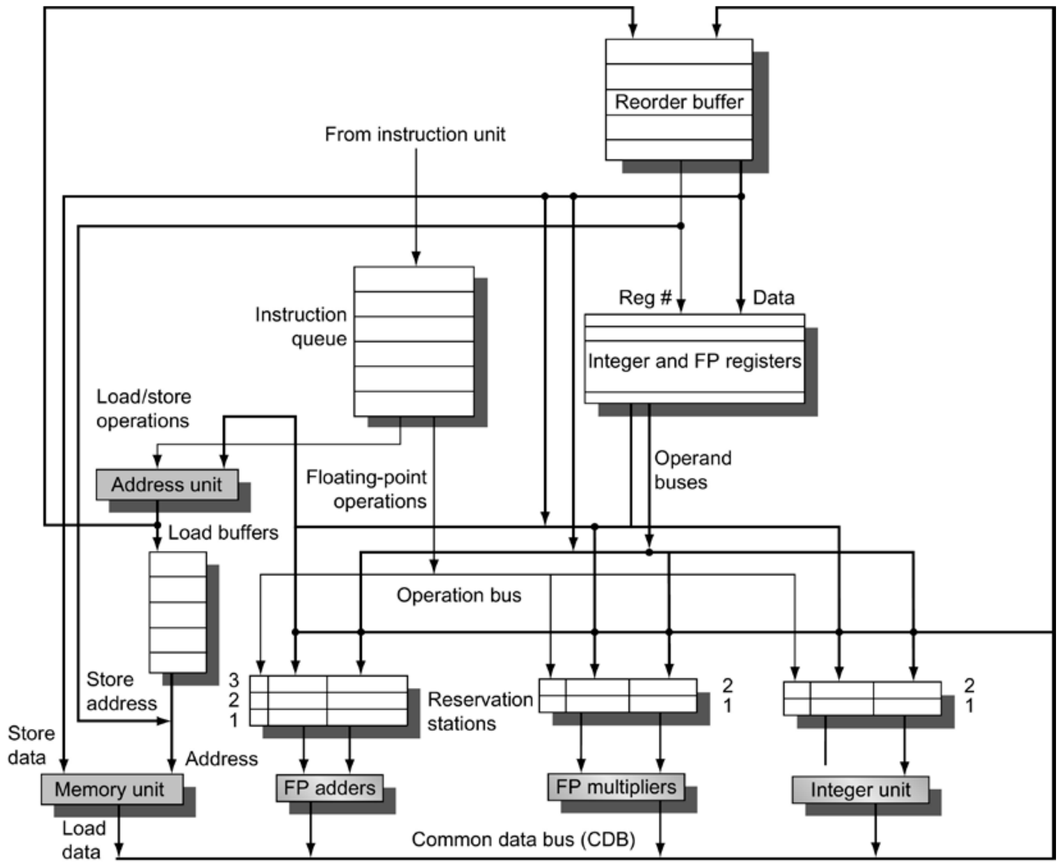


Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

Action or bookkeeping	Comments

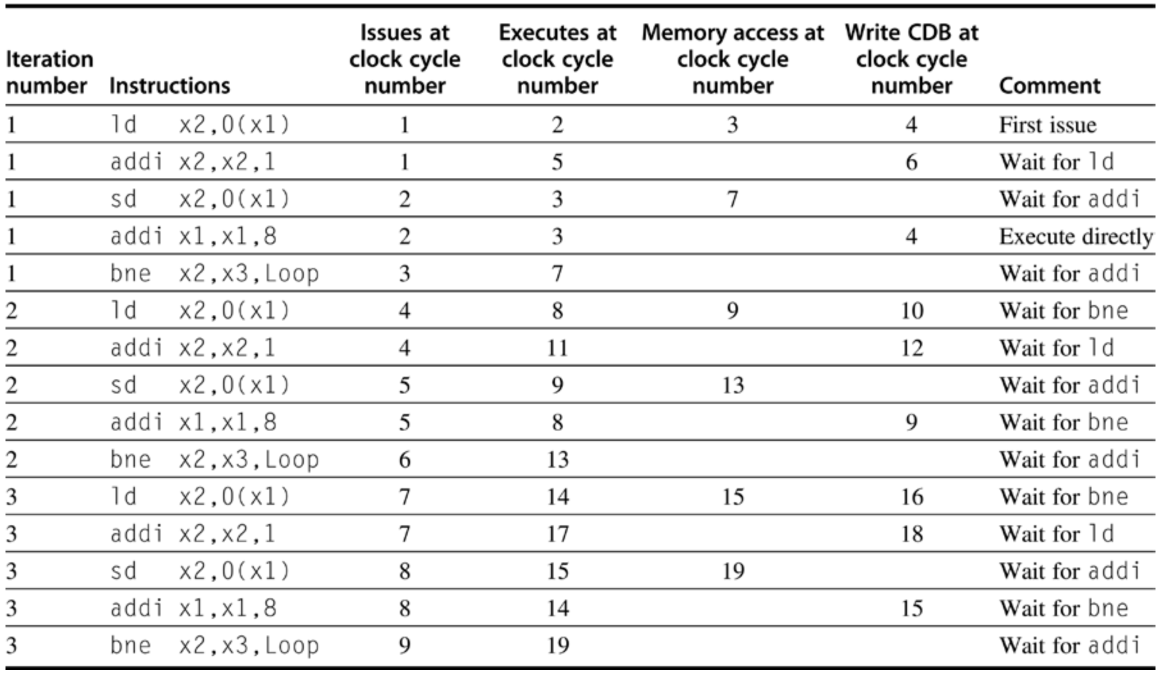
`if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/`
`   {h ← RegisterStat[rs1].Reorder;`
`   if (ROB[h].Ready)/* Instr completed already */`
`      {RS[x1].Vj ← ROB[h].Value; RS[x1].Qj ← 0;}`
`   else {RS[x1].Qj ← h;} /* wait for instruction */`
`} else {RS[x1].Vj ← Regs[rs]; RS[x1].Qj ← 0;};`
`RS[x1].Busy ← yes; RS[x1].Dest ← b1;`
`ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1;`
`ROB[b1].Ready ← no;`
`RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1;`
`RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;`
	Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.

`RS[x2].Qj ← b1;} /* wait for load instruction */`	Because we know that the first operand of the FP operation is from the load, this step simply updates the reservation station to point to the load. Notice that the dependence must be analyzed on the fly and the ROB entries must be allocated during this issue step so that the reservation tables can be correctly updated.

`if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/`
`   {h ← RegisterStat[rt2].Reorder;`
`   if (ROB[h].Ready)/* Instr completed already */`
`      {RS[x2].Vk ← ROB[h].Value; RS[x2].Qk ← 0;}`
`   else {RS[x2].Qk ← h;} /* wait for instruction */`
`} else {RS[x2].Vk ← Regs[rt2]; RS[x2].Qk ← 0;};`
`RegisterStat[rd2].Reorder ← b2;`
`RegisterStat[rd2].Busy ← yes;`
`ROB[b2].Dest ← rd2;`
	Because we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction were dependent on something in the same issue bundle, the tables would need to be updated using the assigned reservation buffer.

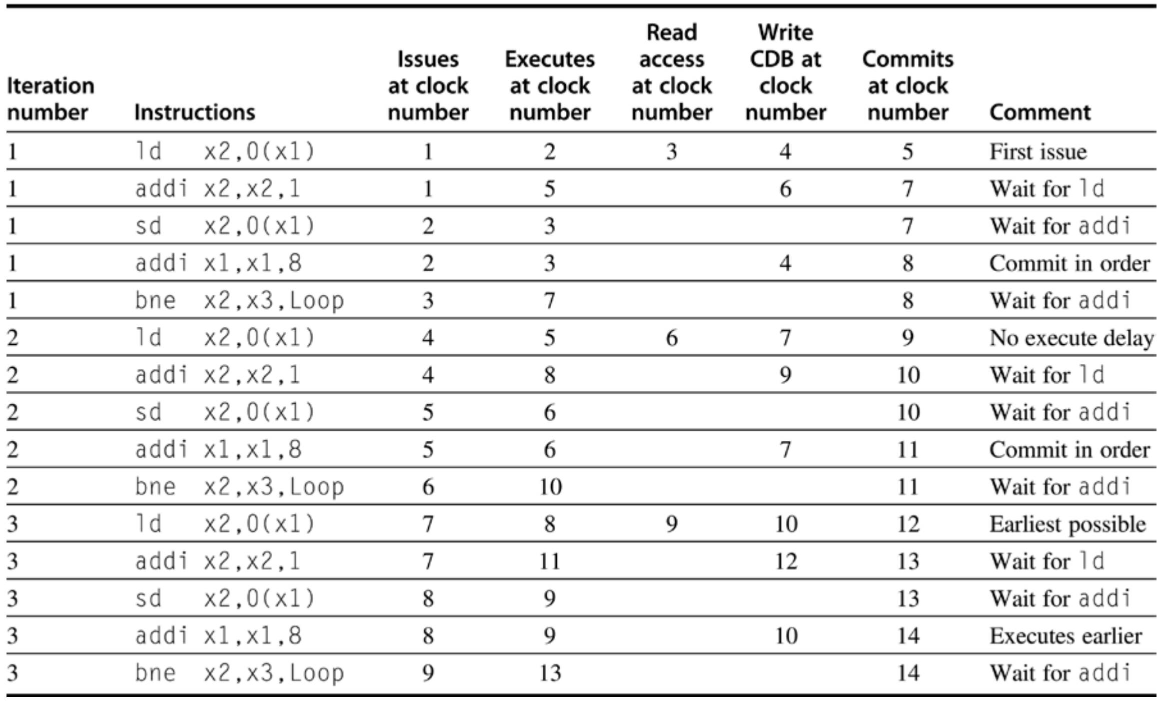
`RS[x2].Busy ← yes; RS[x2].Dest ← b2;`
`ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2;`
`ROB[b2].Ready ← no;`
	This section simply updates the tables for the FP operation and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.

Figure 3.22 The issue steps for a pair of dependent instructions (called 1 and 2), where instruction 1 is FP load and instruction 2 is an FP operation whose first operand is the result of the load instruction; `x1` and `x2` are the assigned reservation stations for the instructions; and `b1` and `b2` are the assigned reorder buffer entries. For the issuing instructions, `rd1` and `rd2` are the destinations; `rs1`, `rs2`, and `rt2` are the sources (the load has only one source); `x1` and `x2` are the reservation stations allocated; and `b1` and `b2` are the assigned ROB entries. RS is the reservation station data structure. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure. Notice that we need to have assigned reorder buffer entries for this logic to operate properly, and recall that all these updates happen in a single clock cycle in parallel, not sequentially.



Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	`ld x2,0(x1)`	1	2	3	4	First issue
1	`addi x2,x2,1`	1	5		6	Wait for `ld`
1	`sd x2,0(x1)`	2	3	7		Wait for `addi`
1	`addi x1,x1,8`	2	3		4	Execute directly
1	`bne x2,x3,Loop`	3	7			Wait for `addi`
2	`ld x2,0(x1)`	4	8	9	10	Wait for `bne`
2	`addi x2,x2,1`	4	11		12	Wait for `ld`
2	`sd x2,0(x1)`	5	9	13		Wait for `addi`
2	`addi x1,x1,8`	5	8		9	Wait for `bne`
2	`bne x2,x3,Loop`	6	13			Wait for `addi`
3	`ld x2,0(x1)`	7	14	15	16	Wait for `bne`
3	`addi x2,x2,1`	7	17		18	Wait for `ld`
3	`sd x2,0(x1)`	8	15	19		Wait for `addi`
3	`addi x1,x1,8`	8	14		15	Wait for `bne`
3	`bne x2,x3,Loop`	9	19			Wait for `addi`

Figure 3.23 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the `ld` following the `bne` cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 3.24 shows this example with speculation.



Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	`ld x2,0(x1)`	1	2	3	4	5	First issue
1	`addi x2,x2,1`	1	5		6	7	Wait for `ld`
1	`sd x2,0(x1)`	2	3			7	Wait for `addi`
1	`addi x1,x1,8`	2	3		4	8	Commit in order
1	`bne x2,x3,Loop`	3	7			8	Wait for `addi`
2	`ld x2,0(x1)`	4	5	6	7	9	No execute delay
2	`addi x2,x2,1`	4	8		9	10	Wait for `ld`
2	`sd x2,0(x1)`	5	6			10	Wait for `addi`
2	`addi x1,x1,8`	5	6		7	11	Commit in order
2	`bne x2,x3,Loop`	6	10			11	Wait for `addi`
3	`ld x2,0(x1)`	7	8	9	10	12	Earliest possible
3	`addi x2,x2,1`	7	11		12	13	Wait for `ld`
3	`sd x2,0(x1)`	8	9			13	Wait for `addi`
3	`addi x1,x1,8`	8	9		10	14	Executes earlier
3	`bne x2,x3,Loop`	9	13			14	Wait for `addi`

Figure 3.24 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the `ld` following the `bne` can start execution early because it is speculative.

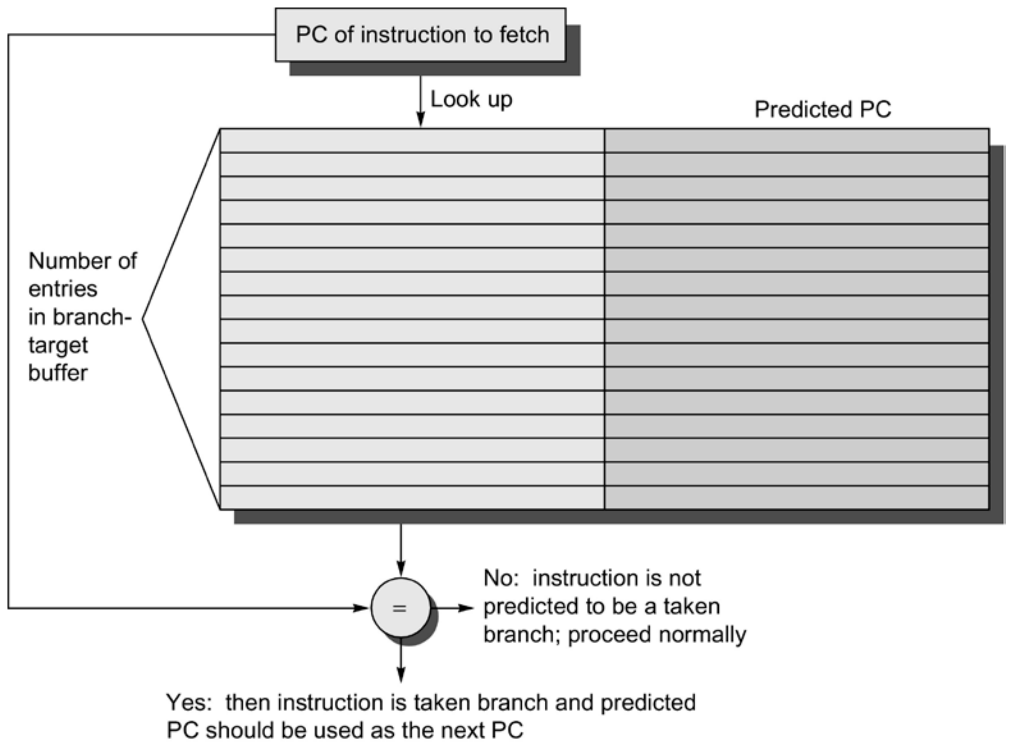

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be a taken branch; proceed normally
Yes: then instruction is taken branch and predicted PC should be used as the next PC

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

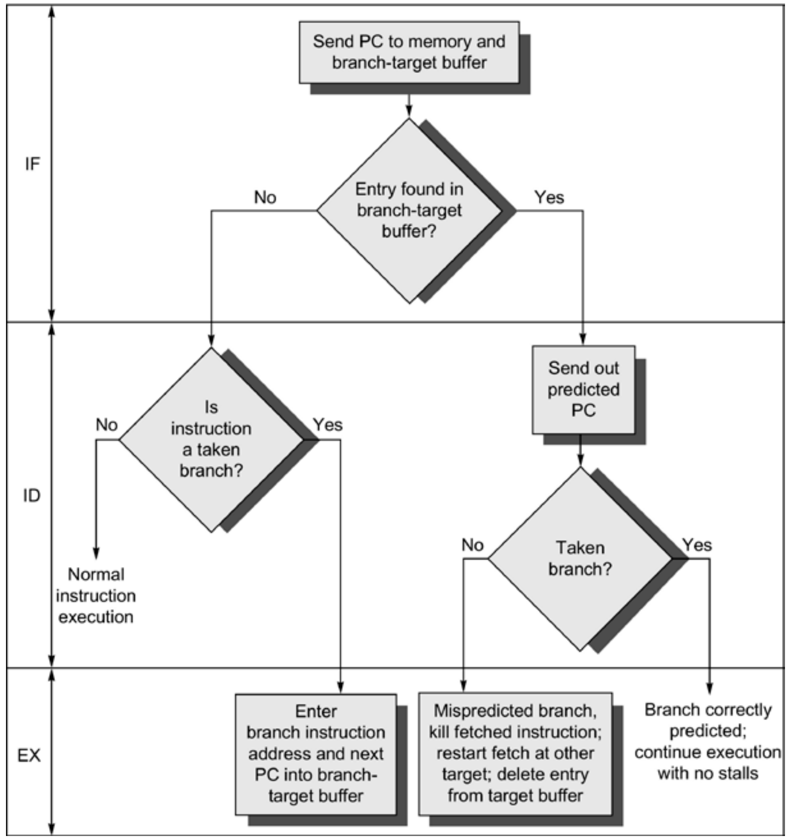
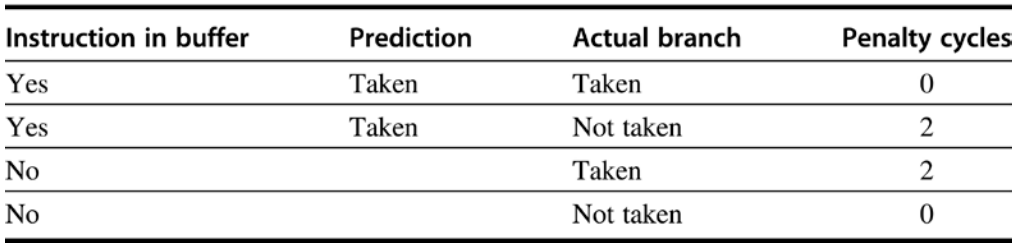

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.



Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

Figure 3.27 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

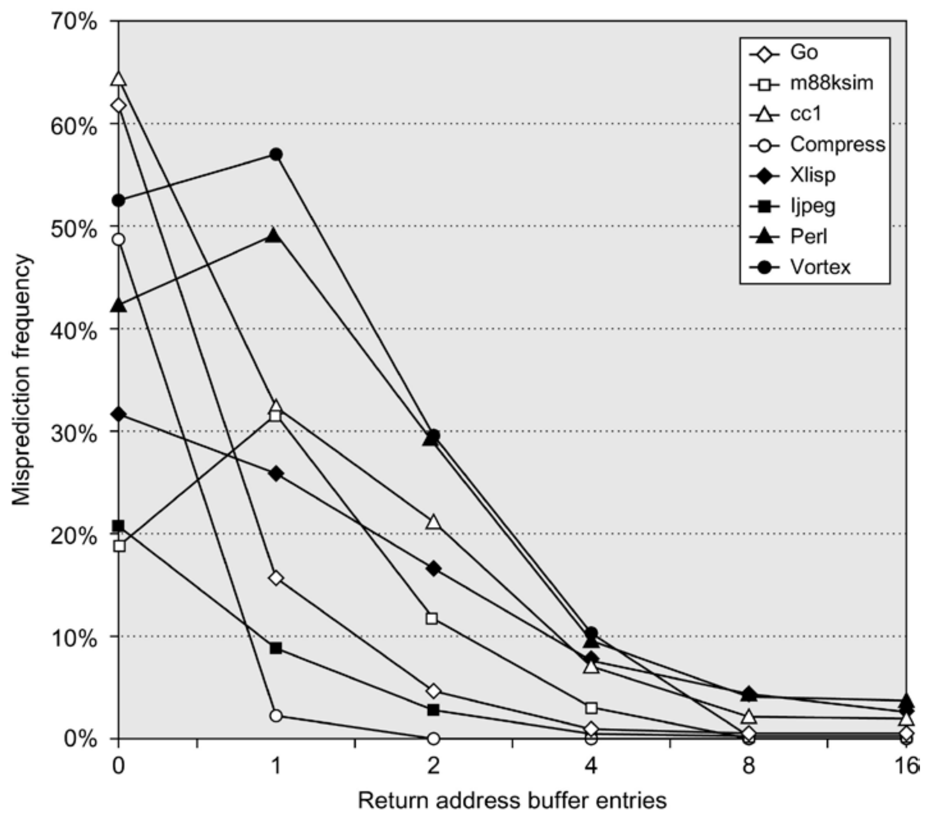


Figure 3.28 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. (1999) and use a fix-up mechanism to prevent corruption of the cached return addresses.

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	`add x1,x2,x3`	p32	`add p32,p2,p3`	**x1-> p32**
2	`sub x1,x1,x2`	p33	`sub p33,p32,p2`	**x1->p33**
3	`add x2,x1,x2`	p34	`add p34,p33,x2`	**x2->p34**
4	`sub x1,x3,x2`	p35	`sub p35,p3,p34`	**x1->p35**
5	`add x1,x1,x2`	p36	`add p36,p35,p34`	**x1->p36**
6	`sub x1,x3,x1`	p37	`sub p37,p3,p36`	**x1->p37**

Figure 3.29 An example of six instructions to be issued in the same clock cycle and what has to happen. The instructions are shown in program order: 1–6; they are, however, issued in 1 clock cycle! The notation `pi` is used to refer to a physical register; the contents of that register at any point is determined by the renaming map. For simplicity, we assume that the physical registers holding the architectural registers `x1, x2,` and `x3` are initially `p1, p2,` and `p3` (they could be any physical register). The instructions are issued with physical register numbers, as shown in column four. The rename map, which appears in the last column, shows how the map would change if the instructions were issued sequentially. The difficulty is that all this renaming and replacement of architectural registers by physical renaming registers happens effectively in 1 cycle, not sequentially. The issue logic must find all the dependences and "rewrite" the instruction in parallel.

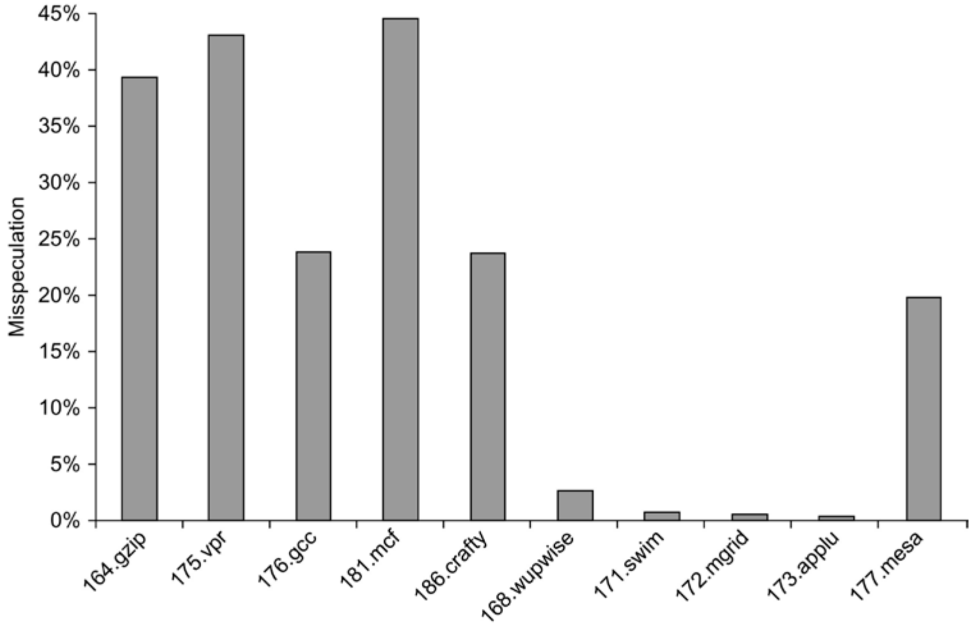


Figure 3.30 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

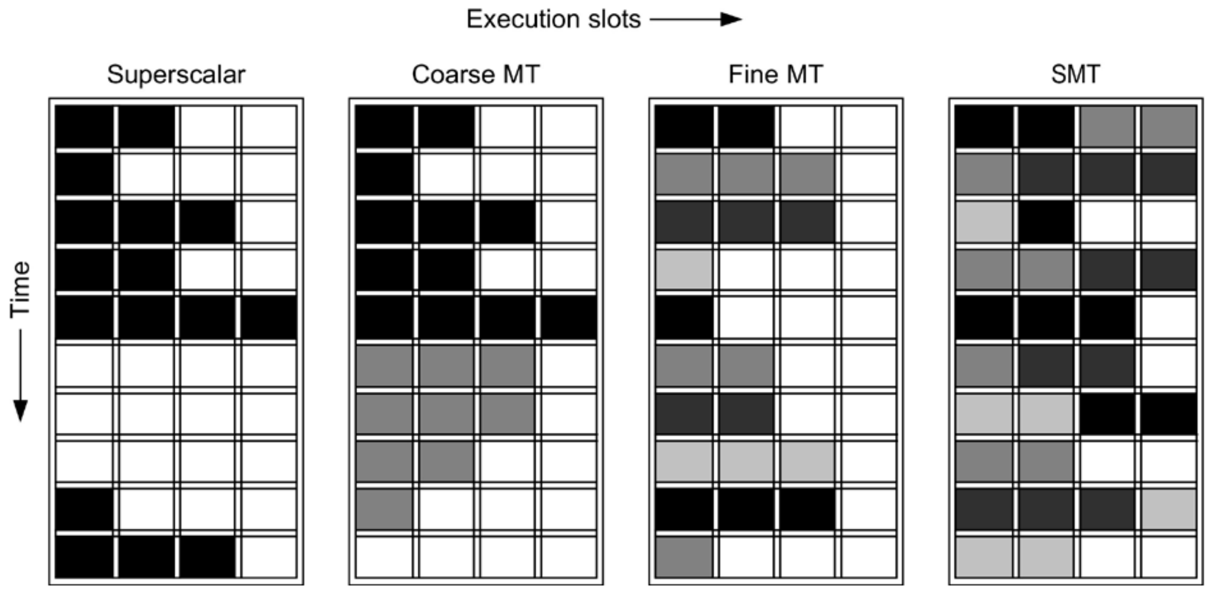
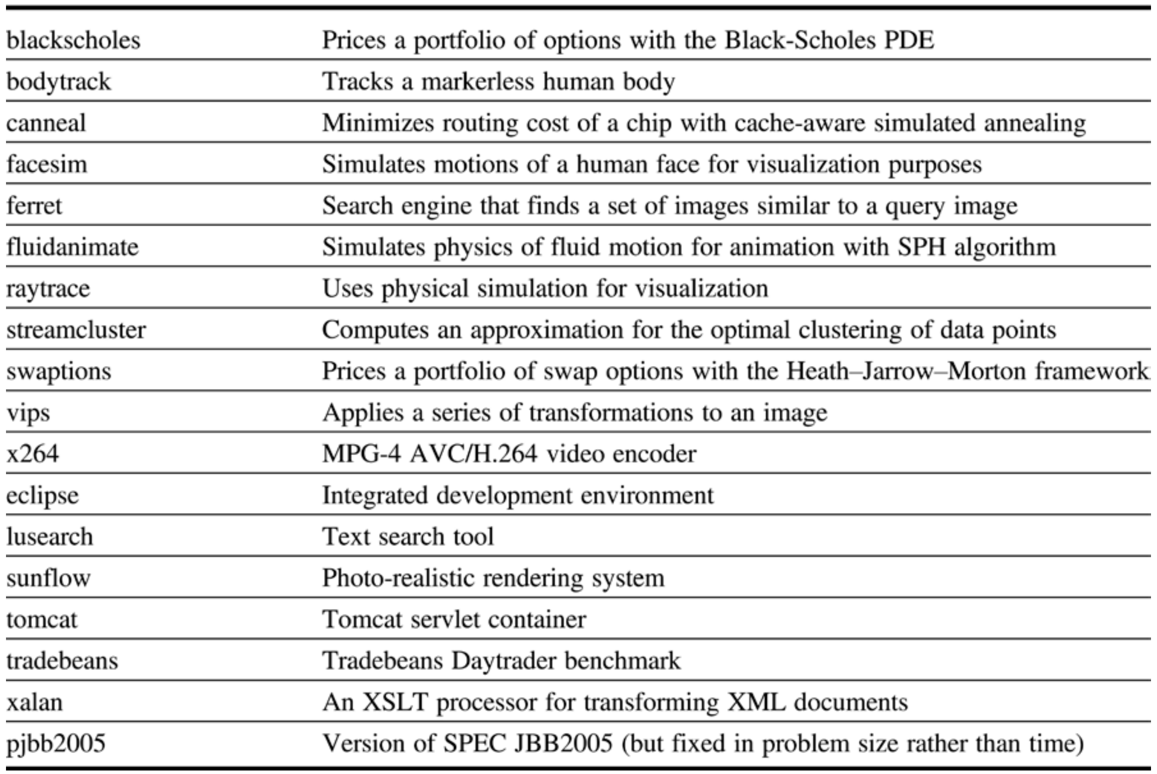


Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.



blackscholes	Prices a portfolio of options with the Black-Scholes PDE
bodytrack	Tracks a markerless human body
canneal	Minimizes routing cost of a chip with cache-aware simulated annealing
facesim	Simulates motions of a human face for visualization purposes
ferret	Search engine that finds a set of images similar to a query image
fluidanimate	Simulates physics of fluid motion for animation with SPH algorithm
raytrace	Uses physical simulation for visualization
streamcluster	Computes an approximation for the optimal clustering of data points
swaptions	Prices a portfolio of swap options with the Heath–Jarrow–Morton framework
vips	Applies a series of transformations to an image
x264	MPG-4 AVC/H.264 video encoder
eclipse	Integrated development environment
lusearch	Text search tool
sunflow	Photo-realistic rendering system
tomcat	Tomcat servlet container
tradebeans	Tradebeans Daytrader benchmark
xalan	An XSLT processor for transforming XML documents
pjbb2005	Version of SPEC JBB2005 (but fixed in problem size rather than time)

Figure 3.32 The parallel benchmarks used here to examine multithreading, as well as in Chapter 5 to examine multiprocessing with an i7. The top half of the chart consists of PARSEC benchmarks collected by Bienia et al. (2008). The PARSEC benchmarks are meant to be indicative of compute-intensive, parallel applications that would be appropriate for multicore processors. The lower half consists of multithreaded Java benchmarks from the DaCapo collection (see Blackburn et al., 2006) and pjbb2005 from SPEC. All of these benchmarks contain some parallelism; other Java benchmarks in the DaCapo and SPEC Java workloads use multiple threads but have little or no true parallelism and, hence, are not used here. See Esmaeilzadeh et al. (2011) for additional information on the characteristics of these benchmarks, relative to the measurements here and in Chapter 5.

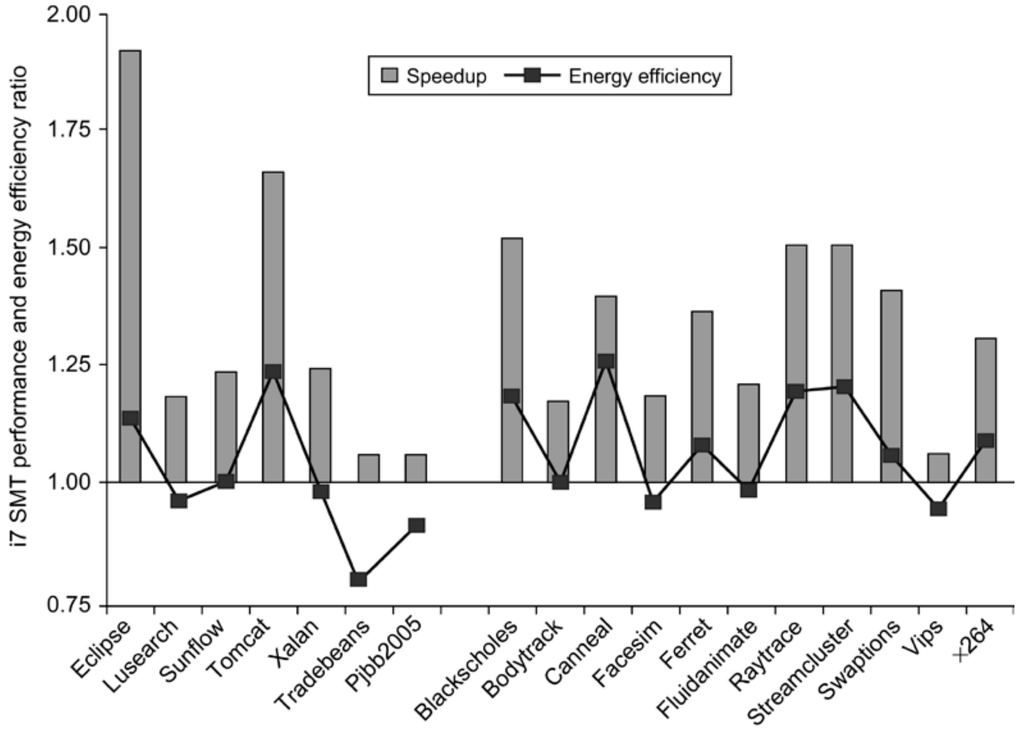


Figure 3.33 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was the same). The energy efficiency averages 0.99 and 1.07, respectively (using the harmonic mean). Recall that anything above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power. Two of the Java benchmarks experience little speedup and have significant negative energy efficiency because of this issue. Turbo Boost is off in all cases. These data were collected and analyzed by Esmaeilzadeh et al. (2011) using the Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine and the gcc v4.4.1 native compiler.

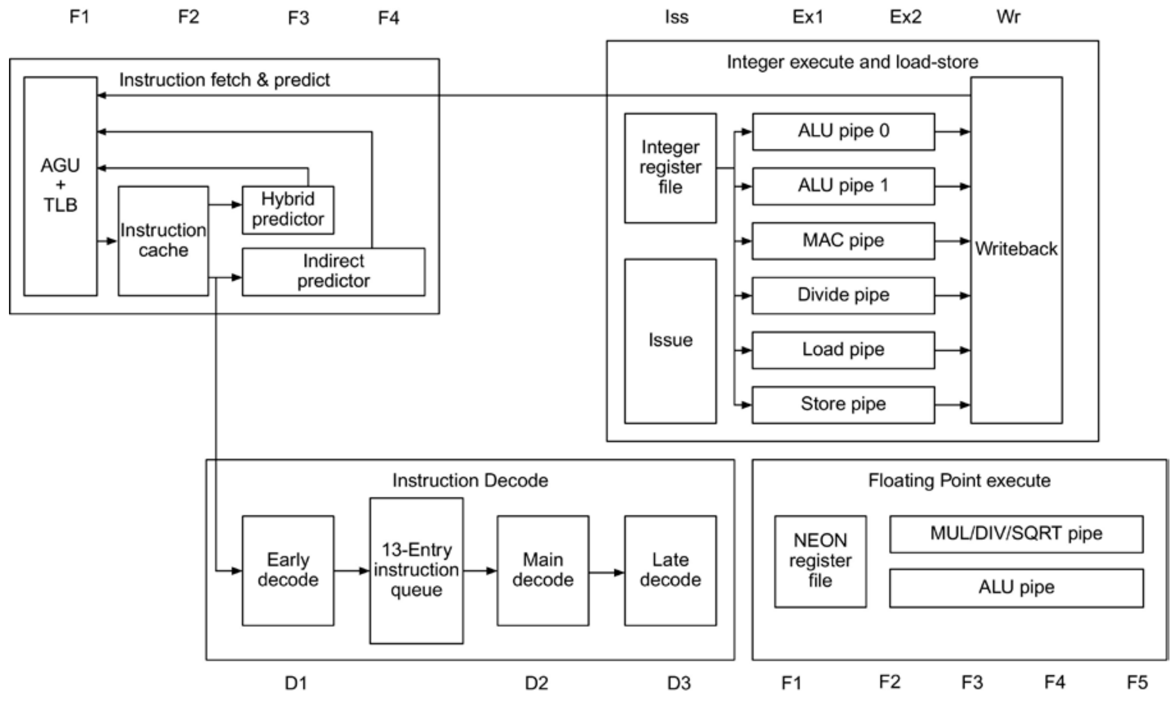


Figure 3.34 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

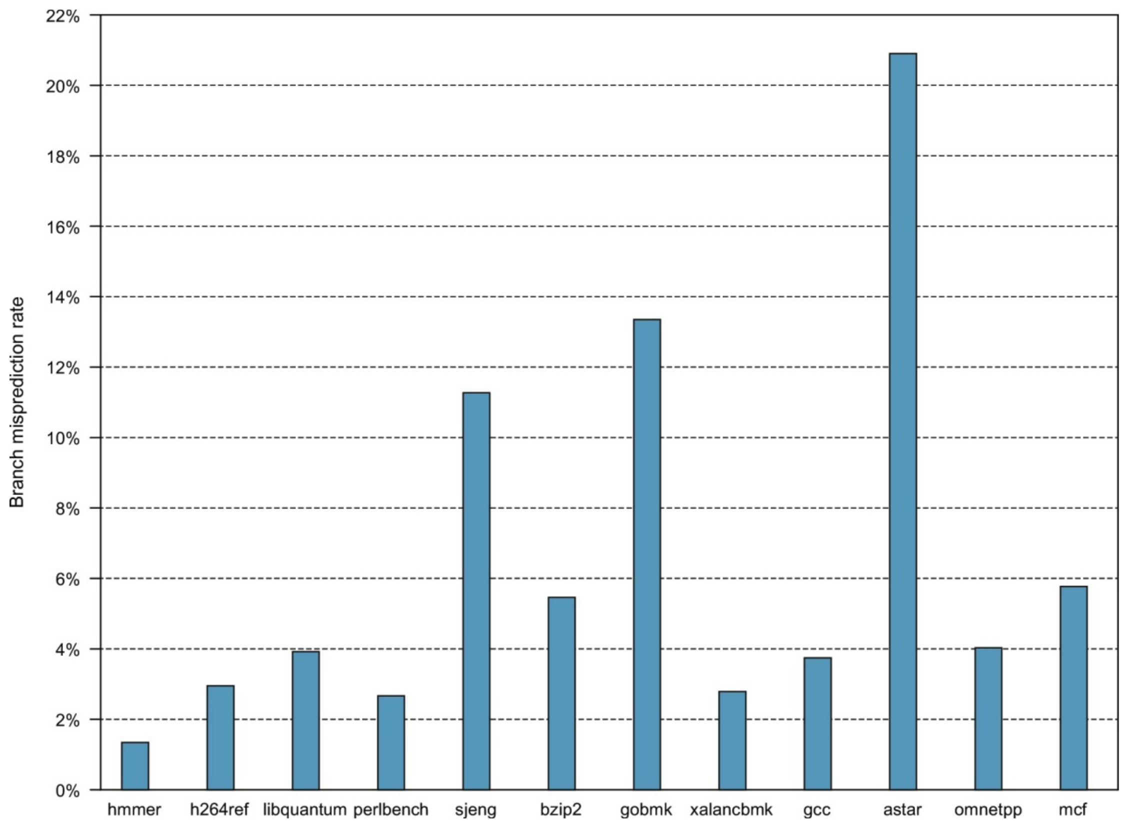


Figure 3.35 Misprediction rate of the A53 branch predictor for SPECint2006.

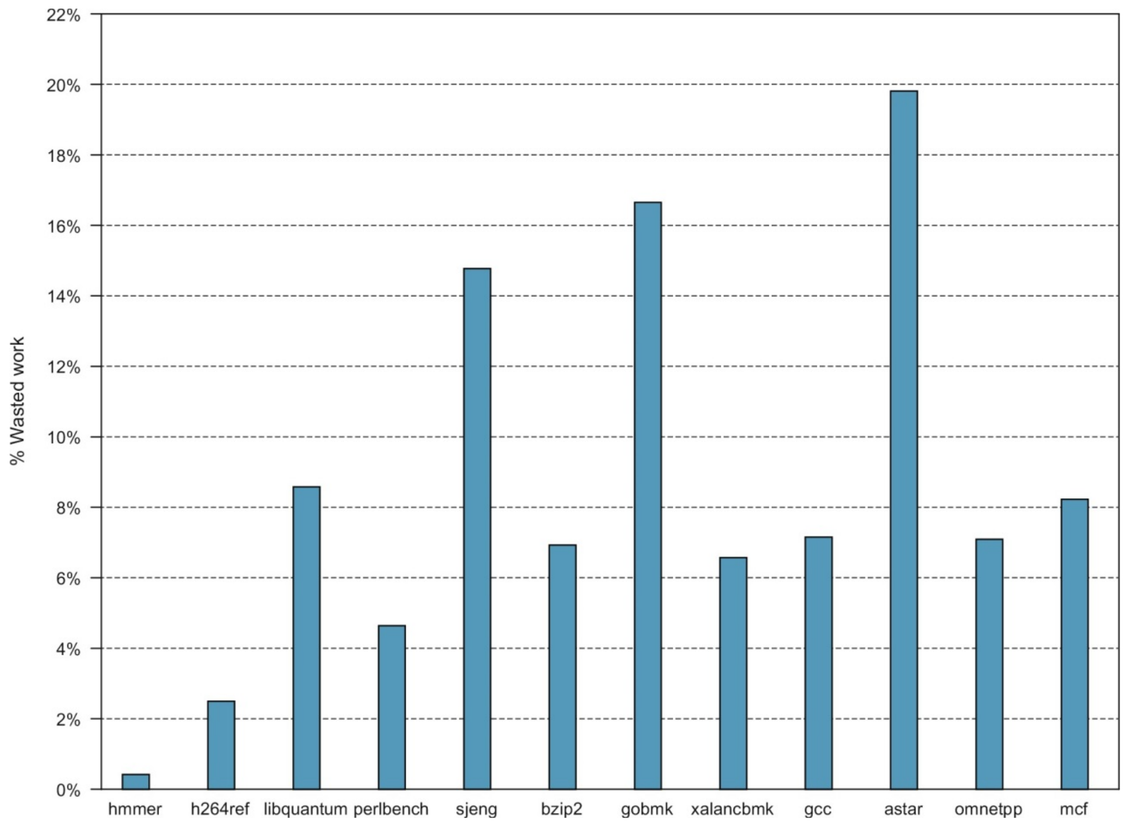


Figure 3.36 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

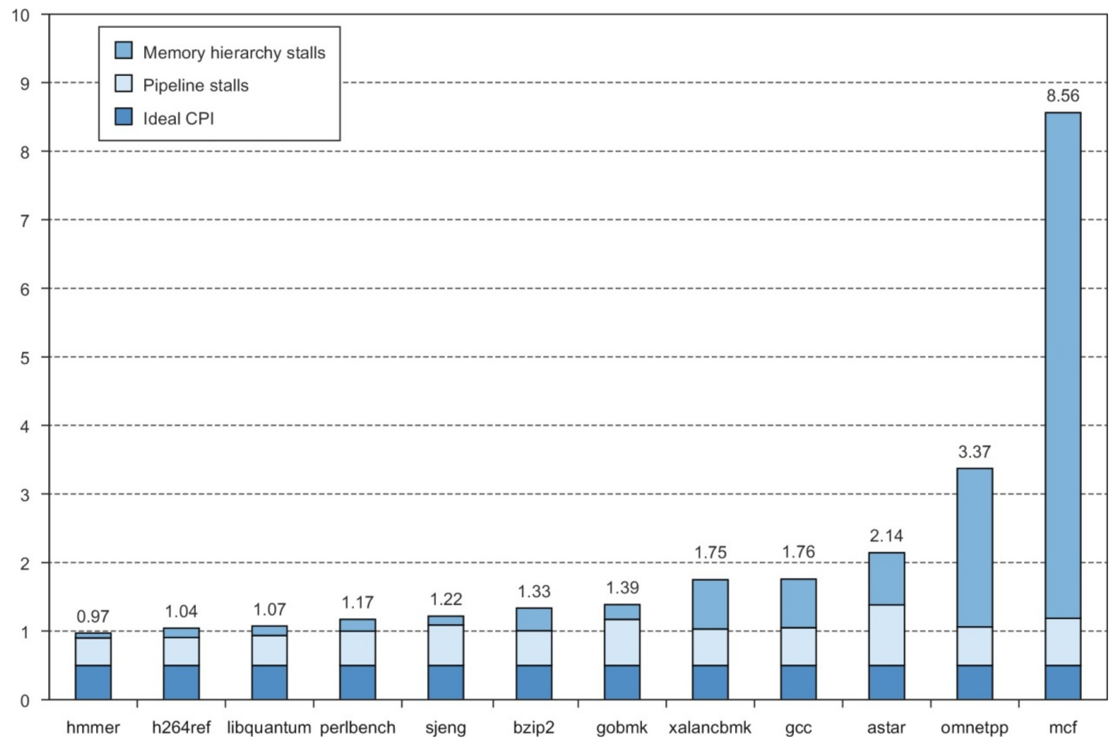


Figure 3.37 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

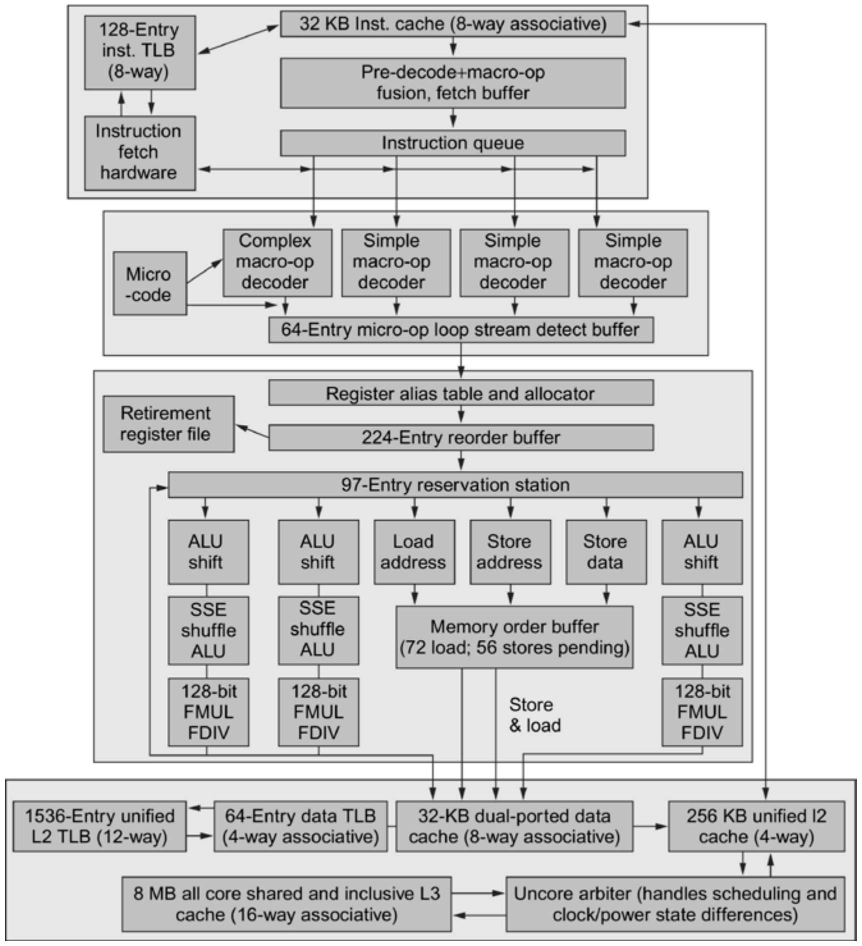

128-Entry inst. TLB (8-way)
32 KB Inst. cache (8-way associative)
Pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
Instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
64-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
224-Entry reorder buffer
97-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer (72 load; 56 stores pending)
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
1536-Entry unified L2 TLB (12-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (4-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Figure 3.38 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles likely due to the time to reset the branch predictor. The six independent functional units can each begin execution of a ready micro-op in the same cycle. Up to four micro-ops can be processed in the register renaming table.

Resource	i7 920 (Nehalem)	i7 6700 (Skylake)
Micro-op queue (per thread)	28	64
Reservation stations	36	97
Integer registers	NA	180
FP registers	NA	168
Outstanding load buffer	48	72
Outstanding store buffer	32	56
Reorder buffer	128	256

Figure 3.39 The buffers and queues in the first generation i7 and the latest generation i7. Nehalem used a reservation station plus reorder buffer organization. In later microarchitectures, the reservation stations serve as scheduling resources, and register renaming is used rather than the reorder buffer; the reorder buffer in the Skylake microarchitecture serves only to buffer control information. The choices of the size of various buffers and renaming registers, while appearing sometimes arbitrary, are likely based on extensive simulation.

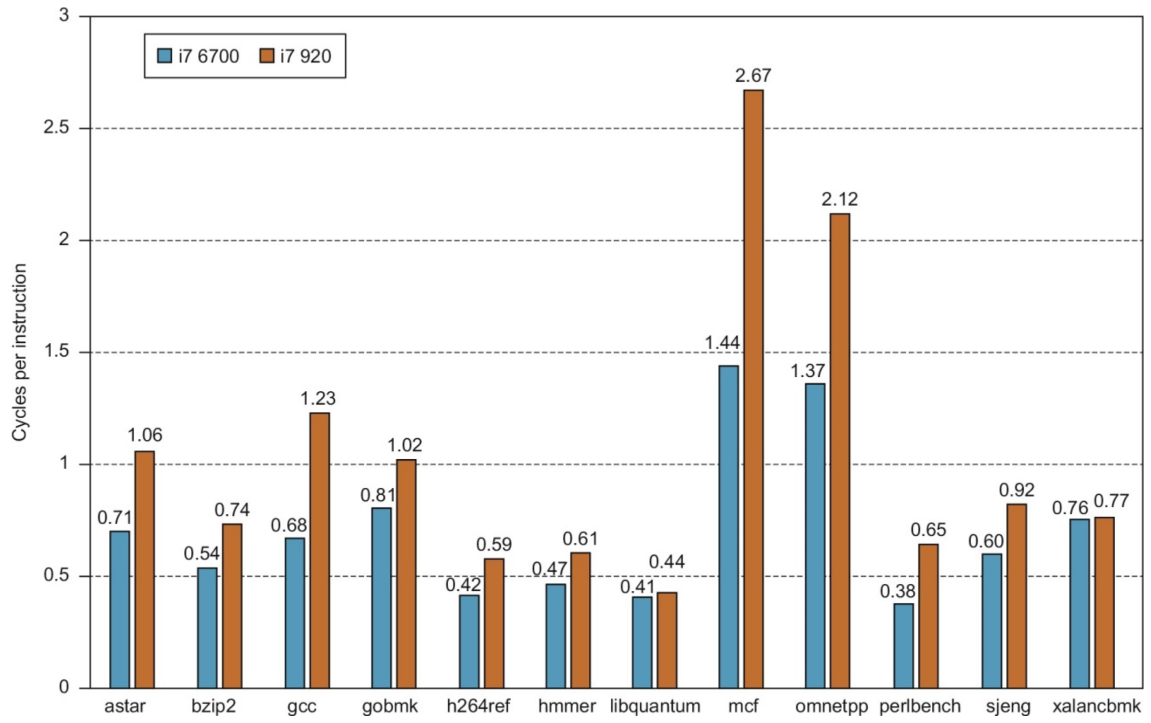
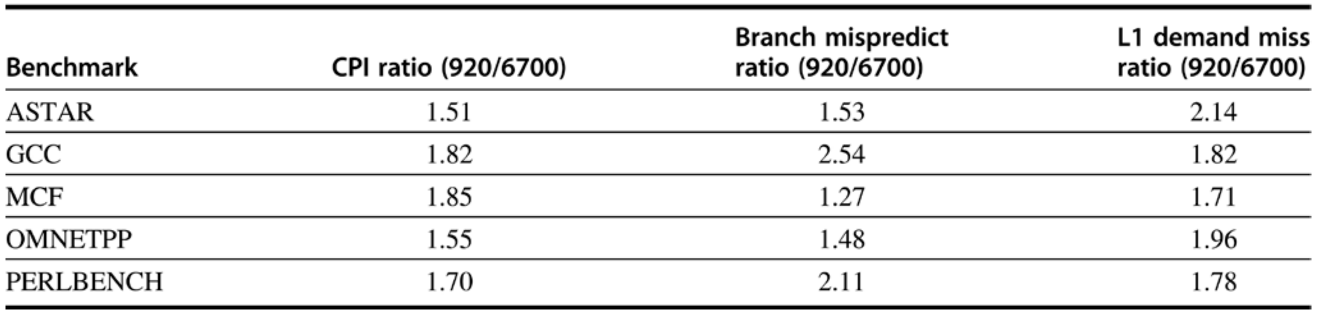
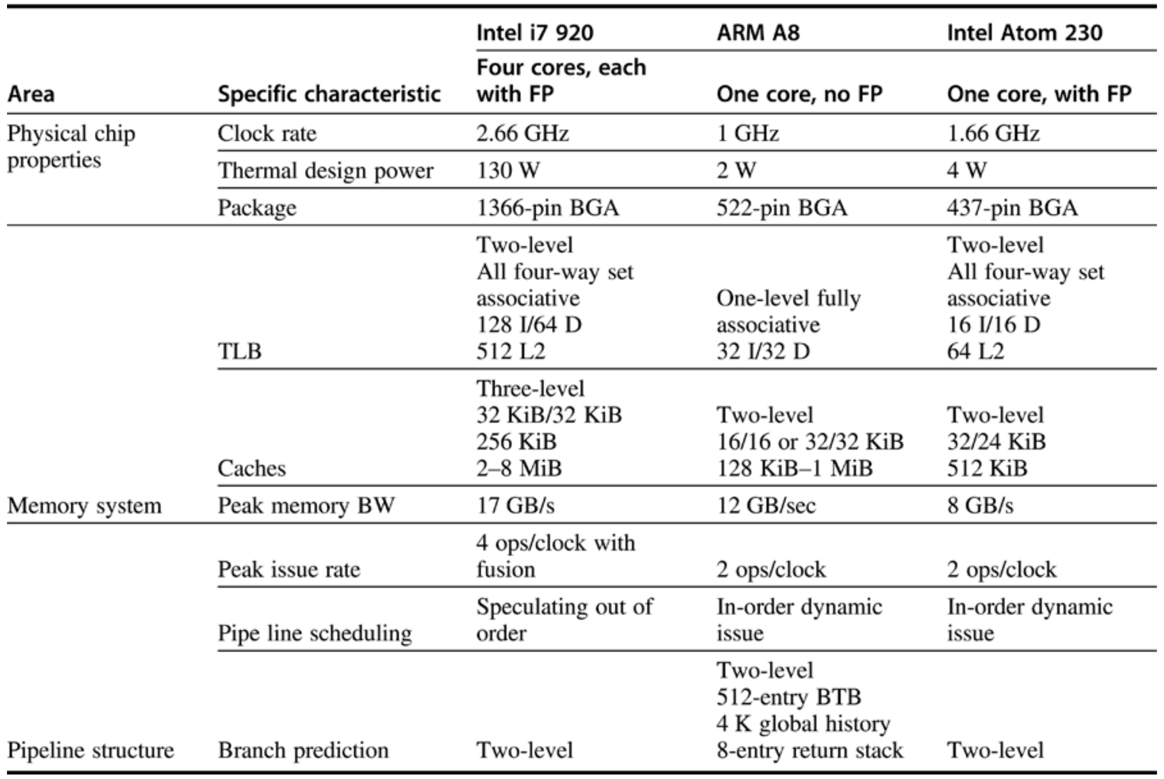


Figure 3.40 The CPI for the SPECCPUint2006 benchmarks on the i7 6700 and the i7 920. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.



Benchmark	CPI ratio (920/6700)	Branch mispredict ratio (920/6700)	L1 demand miss ratio (920/6700)
ASTAR	1.51	1.53	2.14
GCC	1.82	2.54	1.82
MCF	1.85	1.27	1.71
OMNETPP	1.55	1.48	1.96
PERLBENCH	1.70	2.11	1.78

Figure 3.41 An analysis of the five integer benchmarks with the largest performance gap between the i7 6700 and 920. These five benchmarks show an improvement in the branch prediction rate and a reduction in the L1 demand miss rate.



		Intel i7 920	**ARM A8**	**Intel Atom 230**
Area	**Specific characteristic**	**Four cores, each with FP**	**One core, no FP**	**One core, with FP**
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KiB/32 KiB 256 KiB 2–8 MiB	Two-level 16/16 or 32/32 KiB 128 KiB–1 MiB	Two-level 32/24 KiB 512 KiB
	Peak memory BW	17 GB/s	12 GB/sec	8 GB/s
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipe line scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4 K global history 8-entry return stack	Two-level

Figure 3.42 An overview of the four-core Intel i7 920, an example of a typical ARM A8 processor chip (with a 256 MiB L2, 32 KiB L1s, and no floating point), and the Intel ARM 230, clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

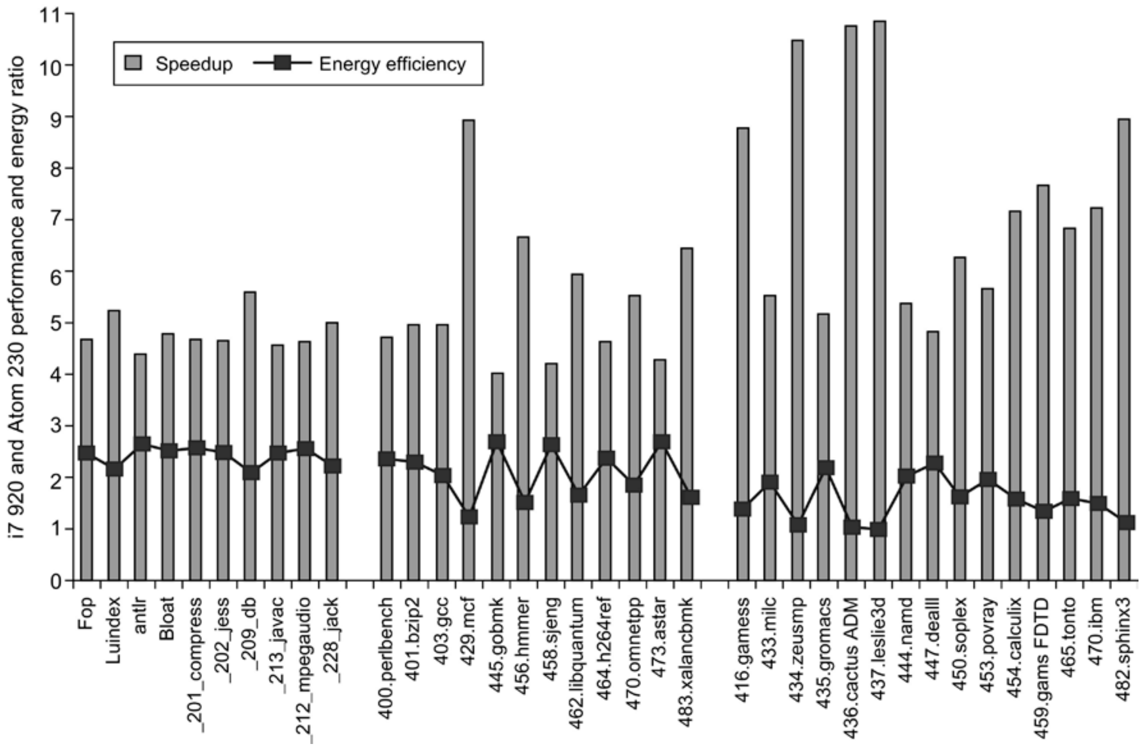
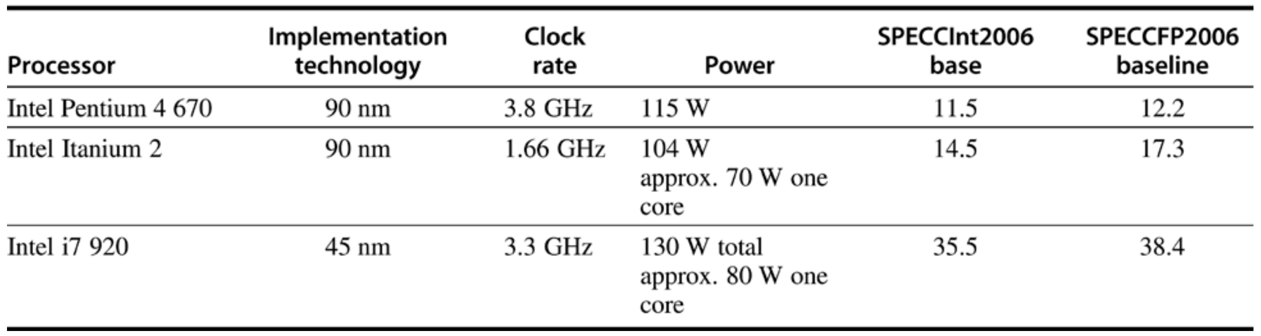


Figure 3.43 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power-efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. The data shown here were collected by Esmaeilzadeh et al. (2011). The SPEC benchmarks were compiled with optimization using the standard Intel compiler, while the Java benchmarks use the Sun (Oracle) Hotspot Java VM. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.



Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

Figure 3.44 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks; the Power column is the thermal design power with estimates for only one core active in the multicore cases.

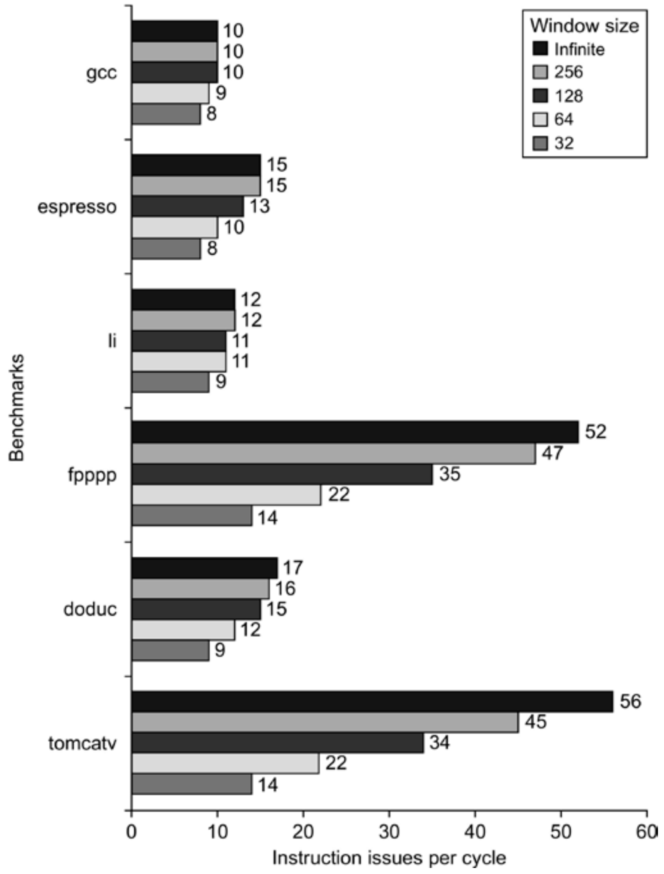
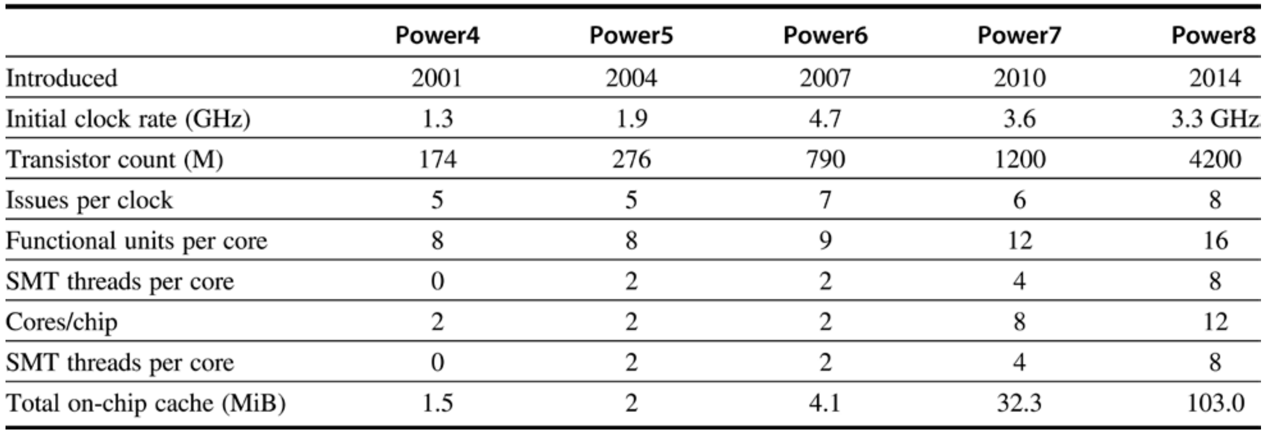


Figure 3.45 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have 1-cycle latency and that the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window.



	Power4	Power5	Power6	Power7	Power8
Introduced	2001	2004	2007	2010	2014
Initial clock rate (GHz)	1.3	1.9	4.7	3.6	3.3 GHz
Transistor count (M)	174	276	790	1200	4200
Issues per clock	5	5	7	6	8
Functional units per core	8	8	9	12	16
SMT threads per core	0	2	2	4	8
Cores/chip	2	2	2	8	12
SMT threads per core	0	2	2	4	8
Total on-chip cache (MiB)	1.5	2	4.1	32.3	103.0

Figure 3.46 Characteristics of five generations of IBM Power processors. All except the Power6, which is static and in-order, were dynamically scheduled; all the processors support two load/store pipelines. The Power6 has the same functional units as the Power5 except for a decimal unit. Power7 and Power8 use embedded DRAM for the L3 cache. Power9 has been described briefly; it further expands the caches and supports off-chip HBM.

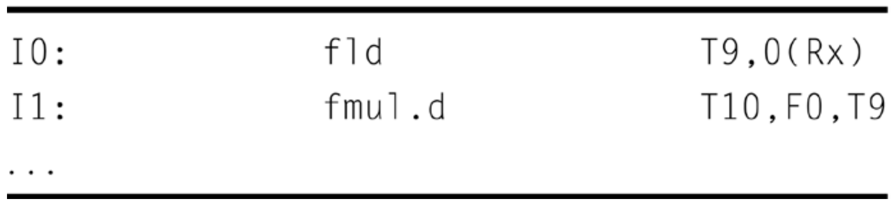
Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
fadd.d	+2
fmul.d	+4
fdiv.d	+10

```
Loop:   fld      f2,0(Rx)
I0:     fmul.d   f2,f0,f2
I1:     fdiv.d   f8,f2,f0
I2:     fld      f4,0(Ry)
I3:     fadd.d   f4,f0,f4
I4:     fadd.d   f10,f8,f2
I5:     fsd      f4,0(Ry)
I6:     addi     Rx,Rx,8
I7:     addi     Ry,Ry,8
I8:     sub      x20,x4,Rx
I9:     bnz      x20,Loop
```

Figure 3.47 Code and latencies for Exercises 3.1 through 3.6.

```
Loop:     fld       f2,0(Rx)
I0:       fmul.d    f5,f0,f2
I1:       fdiv.d    f8,f0,f2
I2:       fld       f4,0(Ry)
I3:       fadd.d    f6,f0,f4
I4:       fadd.d    f10,f8,f2
I5:       sd        f4,0(Ry)
```

Figure 3.48 Sample code for register renaming practice.



```
I0:          fld          T9,0(Rx)
I1:          fmul.d       T10,F0,T9
...
```

Figure 3.49 Expected output of register renaming.

```
I0:    fmul.d    f5,f0,f2
I1:    fadd.d    f9,f5,f4
I2:    fadd.d    f5,f5,f2
I3:    fdiv.d    f2,f9,f0
```

Figure 3.50 Sample code for superscalar register renaming.

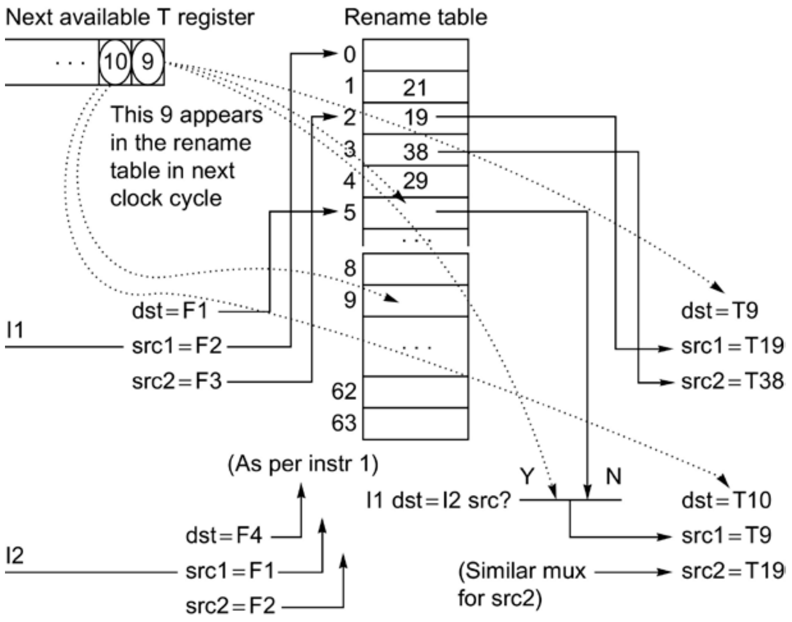


Figure 3.51 Initial state of the register renaming table.

```
Loop:   lw        x1,0(x2);   lw     x3,8(x2)
        <stall>
        <stall>
        addi      x10,x1,1;   addi   x11,x3,1
        sw        x1,0(x2);   sw     x3,8(x2)
        addi      x2,x2,8
        sub       x4,x3,x2
        bnz       x4,Loop
```

Figure 3.52 Sample VLIW code with two adds, two loads, and two stalls.

```
Loop:        lw x1,0(x2)
             addi      x1,x1, 1
             sw        x1,0(x2)
             addi      x2,x2,4
             sub       x4,x3,x2
             bnz       x4,Loop
```

Figure 3.53 Code loop for Exercise 3.11.

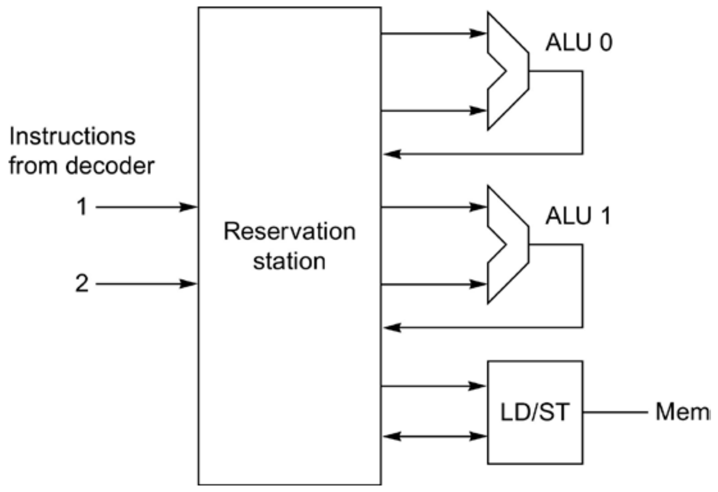


Figure 3.54 Microarchitecture for Exercise 3.12.