Instruction-Level Parallelism and Its Exploitation (Week#14a)

Tomasulo's Algorithm: Details

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 des-

Tomasulo's Algorithm: Details

Figure 3.13 Steps in the algorithm:

- ✓ rd is destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer.
- ✓ RS is the reservation station data structure. 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.
- ✓ Q fields are set to zero either when this instruction is issued.

Tomasulo's Algorithm: Details

- ✓ 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.
- ✓ 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 different during Write Result, where they may have to wait for the value to store.
- ✓ to preserve exception behavior, instructions should not be allowed to execute if a branch in program order has not yet completed.
- ✓ 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.

Tomasulo's Algorithm: A Loop-Based Example

```
Loop:   fld      f0,0(x1)
        fmul.d   f4,f0,f2
        fsd      f4,0(x1)
        addi     x1,x1,−8
        bne      x1,x2,Loop   // branches if x1≠x2
```

		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.

Tomasulo's Algorithm: A Loop-Based Example

A load and a store can be done out of order, provided they access different addresses. If a load and a store access same address then these happens:

1) load is before store in program order and interchanging them results in a WAR hazard.
2) store is before load in program order and interchanging them results in a RAW hazard

In Tomasulo's scheme, two different techniques are combined: the renaming of the architectural registers to a larger set of registers and the buffering of source operands from the register file.

Tomasulo's scheme was adopted in multiple-issue processors for:

1. Designed before caches with unpredictable delays has become motivation for dynamic scheduling. Out-of-order execution allows processors to continue executing instructions while awaiting the completion of a cache miss.

2. When processors became more aggressive in their issue capability, techniques such as register renaming, dynamic scheduling, and speculation became more important.

3. It can achieve high performance without requiring the compiler to target code to a specific pipeline structure

Hardware based Speculation

- ✓ Hardware-based speculation combines three key ideas:
  i. dynamic branch prediction to choose which instructions to execute,
  ii. speculation allow the execution of instructions before the control dependences are resolved (undo effects of an incorrectly speculated sequence)
  iii. dynamic scheduling to deal with scheduling of different combinations of basic blocks.
- ✓ dynamic scheduling without speculation partially overlaps basic blocks because it requires a branch resolved before executing any instructions in successor block.
- ✓ Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions.
- ✓ To extend Tomasulo's algorithm to support speculation, must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively from actual completion of an instruction.
- ✓ By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone.

Hardware based Speculation

Speculation: allow instructions to execute out of order but to force them to commit in order and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.

Re-order buffer (ROB): is a hardware unit used in an extension to Tomasulo's algorithm to support out-of-order and speculative instruction execution. The extension forces instructions to be committed in-order.

- ✓ reorder buffer is also used to pass results among instructions that may be speculated.
- ✓ ROB provides additional registers in the same way as the reservation stations in Tomasulo's algorithm.
- ✓ ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits.
- ✓ ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm.

Hardware based Speculation

Difference between ROB and Reservation Station:

- ✓ In Tomasulo's algorithm, once an instruction writes its result, all subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits
- ✓ ROB supplies operands in the interval between completion of instruction execution and instruction commit. ROB is similar to store buffer in Tomasulo's algorithm, and we integrate the function of the store buffer into the ROB.

Hardware based Speculation

Figure 3.15 shows hardware structure of the processor including the ROB

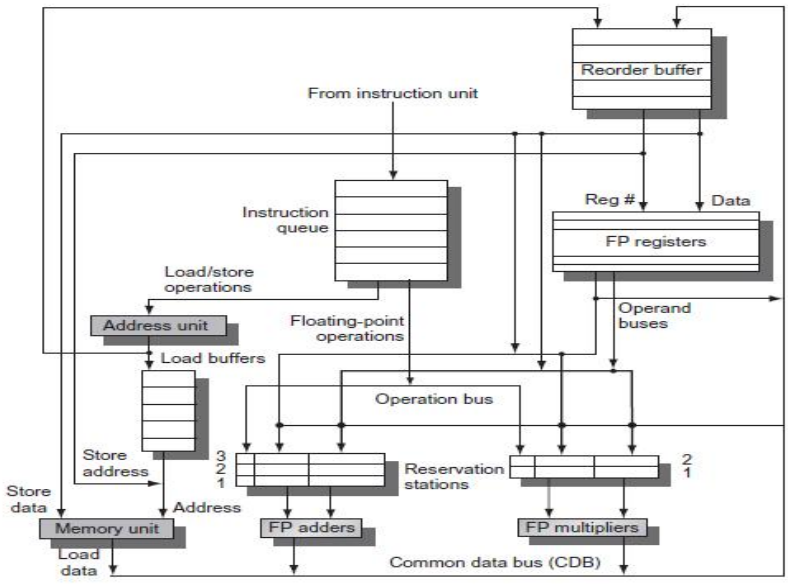

Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.10 on page 198, which implemented Tomasulo's algorithm, we can see that the major change is the addition of the ROB and the elimination of the store buffer, whose function 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.

Hardware based Speculation

- ✓ **Each entry in the ROB** contains four fields: the instruction type, the destination field, the value field, and the ready field.
- ✓ **Instruction type field** indicates whether instruction is a branch (no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations).
- ✓ **Destination field** supplies register number (for loads and ALU operations) or the memory address (for stores) where instruction result should be written.
- ✓ **Value field** is used to hold the value of instruction result until the instruction commits.
- ✓ **Ready field** indicates that instruction has completed execution, and the value is ready.
- ✓ **ROB include the store buffers.** Stores still execute in two steps, but the second step is performed by instruction commit.

Hardware based Speculation

Four steps involved in instruction execution:

1. Issue :

- ✓ Issue instruction if there is an empty reservation station and an empty slot in ROB; send operands to reservation station if they are available.
- ✓ Update the control entries to indicate the buffers are in use.
- ✓ Number of ROB entry allocated for result is also sent to reservation station so that number can be used to tag the result when it is placed on CDB (common data bus).
- ✓ If either all reservations are full or ROB is full, then instruction issue is stalled .

2. Execute:

- ✓ If one or more of operands is not yet available, monitor CDB while waiting for the register to be computed and checks for RAW hazards.
- ✓ When both operands are available at a reservation station, execute the operation.
- ✓ Instructions take multiple clock cycles and loads require two steps
- ✓ Stores only need base register because execution for a store is effective address calculation.

Hardware based Speculation

3. Write result:

- ✓ When result is available, write it on CDB (with ROB tag) and from CDB into ROB and any reservation stations waiting for this result.
- ✓ Mark reservation station as available.If value to be stored is available, it is written into the Value field of the ROB entry for store.
- ✓ If the value to be stored is not available yet, CDB must be monitored until that value is broadcast, at which time Value field of ROB is updated.
- ✓ Assume this occurs during Write Result stage of a store

4. Commit:

- ✓ Commit depend on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit).
- ✓ Normal commit occurs when an instruction reaches head of ROB and its result is present in the buffer;processor updates the register and removes instruction from ROB
- ✓ Committing a store is similar except that memory is updated rather a register.
- ✓ When a branch with incorrect prediction reaches head of ROB, it indicates that speculation was wrong. ROB is flushed and execution is restarted. If branch was correctly predicted,branch is finished

Hardware based Speculation

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the following code segment, the same one we used to generate Figure 3.12, show what the status tables look like when the `fmul.d` is ready to go to commit.

```
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
```

Answer Figure 3.16 shows the result in the three tables. Notice that although the `fsub.d`

Hardware based Speculation

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.

Hardware based Speculation

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 3.14 in execution:

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop    //branches if x1≠x2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the `fld` and `fmul.d` from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (`x1` in this example) and the value (`f4` in this example). Because we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer Figure 3.17 shows the result in two tables.

Hardware based Speculation

	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.

Hardware based Speculation

Status	Wait until	Action or bookkeeping
Issue all instructions		`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;};`
	Reservation station (r) and ROB (b) both available	`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

Hardware based Speculation

- ✓ Figure 3.18 requires store to wait in Write Result stage for register source operand whose value is to be stored; value is then moved from Vk field of store's reservation station to Value field of store's ROB entry.
- ✓ Value to be stored not arrive before store commits and can be placed directly into store's ROB entry by sourcing instruction.
- ✓ This is accomplished by having the hardware track when source value to be stored is available in ROB entry and searching ROB on every instruction.
- ✓ RAW hazards through memory are maintained by two restrictions:
  i. Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load
  ii. Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores
- ✓ these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data.

Exploiting ILP Using Multiple Issue and Static Scheduling

- ✓ The goal of the multiple-issue processors is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three flavors:
- ✓ 1. Statically scheduled superscalar processors
- ✓ 2. VLIW (very long instruction word) processors
- ✓ 3. Dynamically scheduled superscalar processors

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.

Exploiting ILP Using Multiple Issue and Static Scheduling

Basic VLIW Approach: Designed to exploit instruction-level parallelism (ILP).

- ✓ VLIW processor allows programs to explicitly specify instructions to execute in parallel.
- ✓ VLIW is intended to allow higher performance without complexity.
- ✓ VLIW method depends on programs providing all the decisions regarding which instructions to execute simultaneously and how to resolve conflicts.
- ✓ In superscalar designs, number of execution units is invisible to the instruction set. Each instruction encodes one operation only. For most superscalar designs, instruction width is 32 bits or fewer.
- ✓ one VLIW instruction encodes multiple operations, at least one operation for each execution unit of a device.
- ✓ For example, if a VLIW device has five execution units, then a VLIW instruction for the device has five operation fields, each field specifying what operation should be done on that corresponding execution unit.
- ✓ To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and far wider on some architectures.

Exploiting ILP Using Multiple Issue and Static Scheduling

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop `x[i] = x[i] + s` (see page 158 for the RISC-V code) for such a processor. Unroll as many times as necessary to eliminate any stalls.

Answer Figure 3.20 shows the code. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles for the unrolled and scheduled loop. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 3.2 that used unrolled and scheduled code.

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.

Exploiting ILP Using Multiple Issue and Static Scheduling

- ✓ For the original VLIW model, there were both technical and logistical problems that made the approach less efficient.
- ✓ Technical problems were increase in code size and limitations of lockstep operation.
- ✓ Two different elements combine to increase code size for a VLIW.
  i. generating enough operations in a straight-line code fragment requires unrolling loops, thereby increasing code size.
  ii. whenever instructions are not full, unused functional units translate to wasted bits in instruction encoding.
- ✓ Software scheduling approaches such as software pipelining that can achieve benefits of unrolling without code expansion.
- ✓ For example, only one large immediate field for use by any functional unit. Technique is to compress the instructions in main memory and expand them when they are read into cache or decoded.
- ✓ Early VLIWs operated in lockstep; there was no hazard-detection hardware at all.

Exploiting ILP Using Multiple Issue and Static Scheduling

- ✓ Stall in any functional unit pipeline must cause entire processor to stall.
- ✓ Compiler able to schedule deterministic functional units to prevent stalls, predicting which data accesses encounter a cache stall and scheduling them were very difficult .
- ✓ Caches needed to be blocking and causing all the functional units to stall.
- ✓ In more recent processors, functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution.
- ✓ Binary code compatibility is a major logistical problem for general purpose VLIW.
- ✓ Code sequence makes use of both instruction set and pipeline structure, including both functional units and their latencies.
- ✓ Different numbers of functional units and unit latencies require different versions of the code.
- ✓ improved performance from a new superscalar design may require recompilation.

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

- ✓ Tomasulo's algorithm to support multipleissue superscalar pipeline with separate integer, load/store, and floating-point units (both FP multiply and FP add), each of which can initiate an operation on every clock.
- ✓ do not want to issue instructions to the reservation stations out of order because this could lead to a violation of the program semantics.
- ✓ To gain the full advantage of dynamic scheduling, we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit.
- ✓ interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution.

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

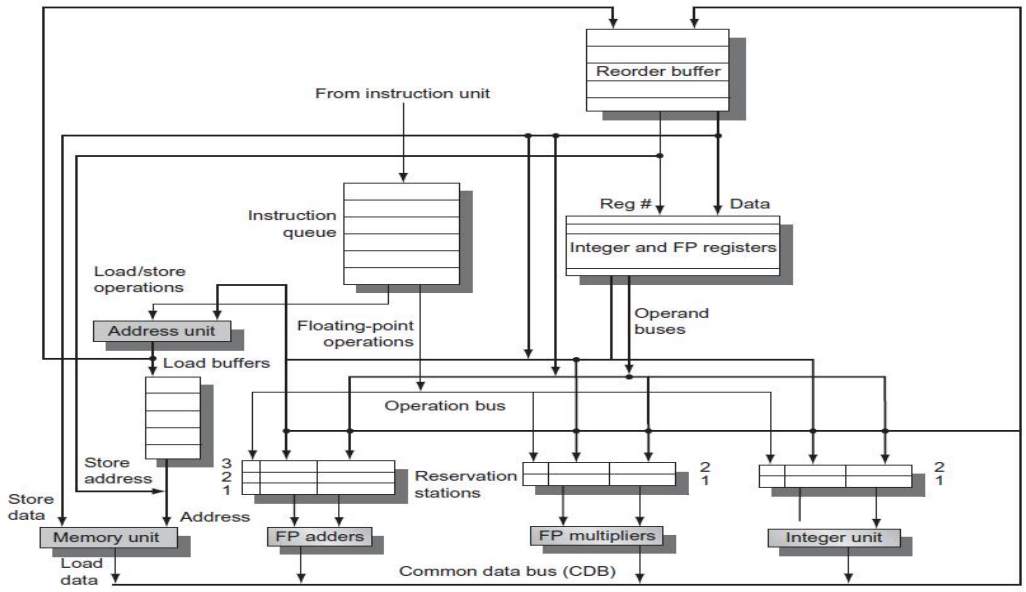

Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to 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.

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

- ✓ Issuing multiple instructions per clock in a dynamically scheduled processor (with or without speculation) is very complex due to multiple instructions may depend on one another.
- ✓ Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor and both rely on assigning a reservation station and updating the pipeline control tables.
  - i. run this step in half clock cycle so that two instructions can be processed in one clock cycle; this approach cannot be easily extended to handle four instructions per clock
  - ii. build the logic necessary to handle two or more instructions at once including any possible dependences between the instructions.
- ✓ By making instruction issues take multiple clocks because new instructions are issuing every clock cycle, assign the reservation station and to update the pipeline tables so that a dependent instruction issuing on the next clock can use the updated information.

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.

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

Generalize the detail of Fig 3.22 to describe the basic strategy for updating issue logic and the reservation tables in a dynamically scheduled superscalar with up to n issues per clock as follows:

1. Assign a reservation station and a reorder buffer for every instruction that might be issued in next issue bundle. This assignment can be done before the instruction types are known simply by preallocating reorder buffer entries sequentially to instructions using n available reorder buffer entries and ensuring that enough reservation stations are available. If sufficient reservation stations not be available,bundle is broken and only a subset of instructions in original program order is issued.
2. Analyze all the dependences among the instructions in the issue bundle.
3. If an instruction in the bundle depends on an earlier instruction in the bundle, use assigned reorder buffer number to update the reservation table for dependent instruction.

Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:  ld    x2,0(x1)      //x2=array element
       addi  x2,x2,1       //increment x2
       sd    x2,0(x1)      //store result
       addi  x1,x1,8       //increment pointer
       bne   x2,x3,Loop    //branch if not last
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.23 and 3.24 show the performance for a two-issue, dynamically scheduled processor, without and with speculation. In this case, where a branch

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.

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and 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.

Advanced Techniques for Instruction Delivery and Speculation

Increasing Instruction Fetch Bandwidth:

- ✓ A multiple-issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput.
- ✓ fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches.

Branch-Target Buffers:

- ✓ To reduce the branch penalty for five-stage pipeline and deeper pipelines, must know whether the instruction is a branch.
- ✓ If the instruction is a branch and know PC, branch penalty will be zero.
- ✓ **Branch target Buffer:** A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.

Advanced Techniques for Instruction Delivery and Speculation

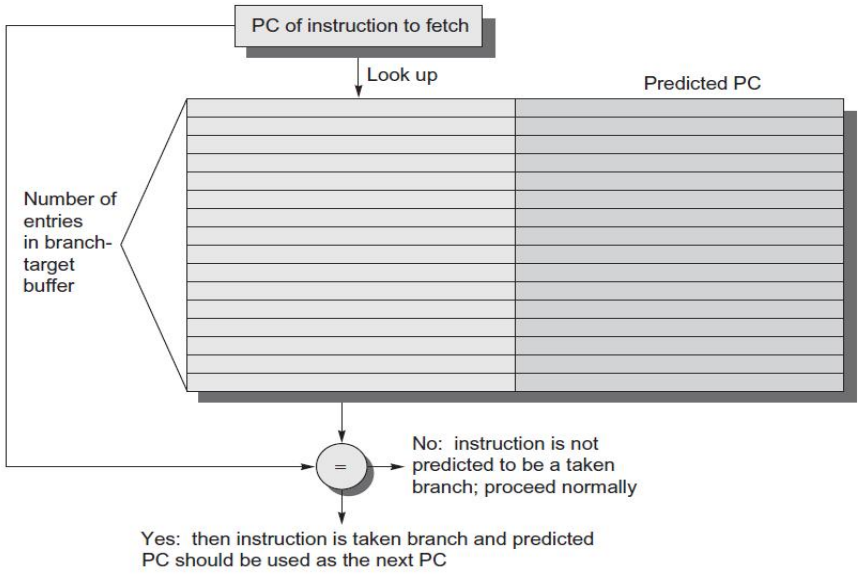

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and 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.

Advanced Techniques for Instruction Delivery and Speculation

- ✓ Fig 3.25 shows that Branch-target buffer predicts next instruction address and send it before decoding the instruction, must know whether the fetched instruction is predicted as a taken branch.
- ✓ If PC of the fetched instruction matches an address in the prediction buffer, then the corresponding predicted PC is used as the next PC.
- ✓ Hardware for this branch-target buffer is identical to the hardware for a cache.
- ✓ If a matching entry is found in branch-target buffer fetching begins immediately at predicted PC.
- ✓ predictive entry must be matched to this instruction because the predicted PC is sent before it is known whether the instruction is even a branch.
- ✓ If the processor did not check whether the entry matched this PC, then the wrong PC would be sent resulting in worse performance.
- ✓ Store only predicted-taken branches in the branch-target buffer because an untaken branch should fetch next sequential instruction.

Advanced Techniques for Instruction Delivery and Speculation

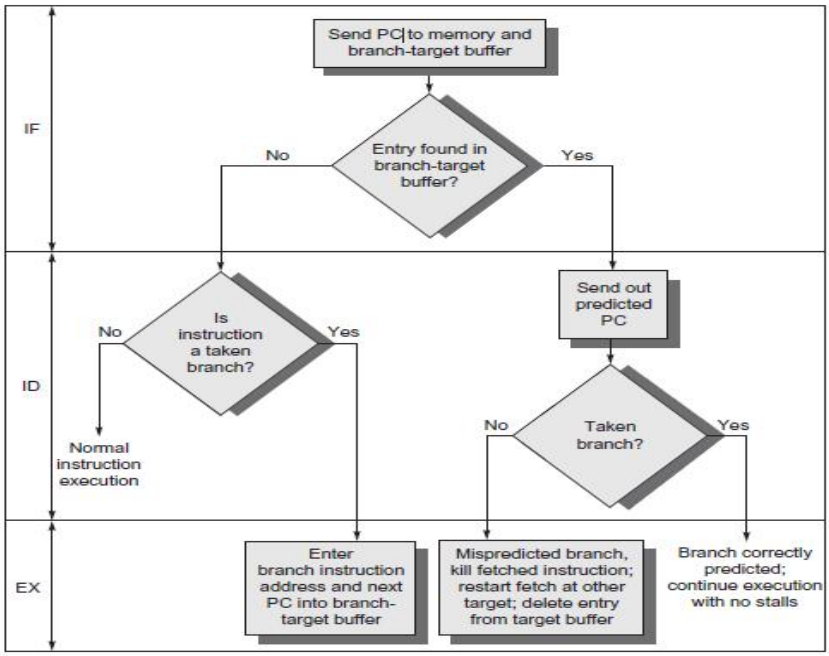

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.

Advanced Techniques for Instruction Delivery and Speculation

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions in Figure 3.27. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

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.

Advanced Techniques for Instruction Delivery and Speculation

Answer We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.

$$\begin{aligned}
\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \\
&\quad \times \text{Percent incorrect predictions} \\
&= 90\% \times 10\% = 0.09 \\
\text{Probability (branch not in buffer, but actually taken)} &= 10\% \\
\text{Branch penalty} &= (0.09 + 0.10) \times 2 \\
\text{Branch penalty} &= 0.38
\end{aligned}$$

The improvement from dynamic branch prediction will grow as the pipeline length, and thus the branch delay grows; in addition, better predictors will yield a greater performance advantage. Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

Advanced Techniques for Instruction Delivery and Speculation

- ✓ Variation on branch-target buffer is to store one or more target instructions instead of predicted target address.
- ✓ This variation has two potential advantages.
  - i. allows branch-target buffer access takes longer time possibly allowing a larger branch-target buffer.
  - ii. buffering actual target instructions allows to perform an optimization called branch folding. Branch folding can be used to obtain 0-cycle unconditional branches and 0-cycle conditional branches.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

- ✓ Procedure returns can be predicted with a branch-target buffer, accuracy of such a prediction technique can be low if the procedure is called from multiple sites and the calls from one site are not clustered in time.
- ✓ For example,in SPEC CPU95, an aggressive branch predictor achieves an accuracy of less than 60% for such return branches.

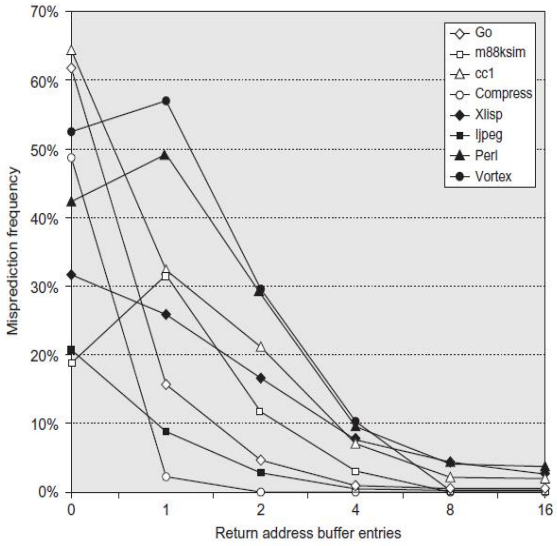

Figure 3.28 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not 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.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

- ✓ To overcome this problem, some designs use a small buffer of return addresses operating as a stack.
- ✓ This structure caches the most recent return addresses, pushing a return address on the stack at a call and popping one off at a return.
- ✓ If the cache is sufficiently large (as large as the maximum call depth), it will predict the returns perfectly.

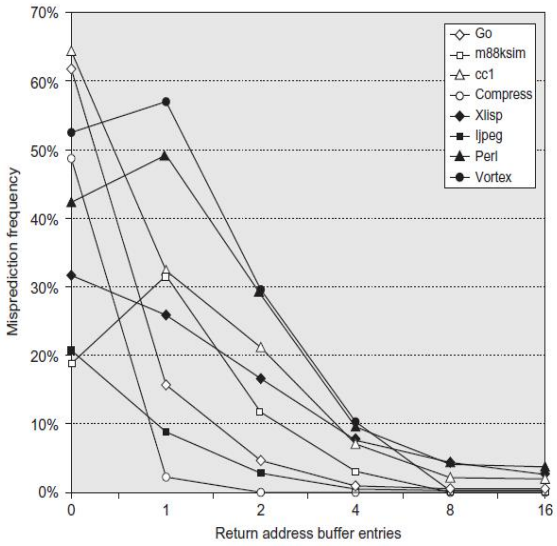

Figure 3.28 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not 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.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

Integrated Instruction Fetch Units: Recent designs used integrated instruction fetch unit that integrates several functions:

1. **Integrated branch prediction:** Branch predictor becomes part of the instruction fetch unit and is constantly predicting branches .
2. **Instruction prefetch:** To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions integrating it with branch prediction.
3. **Instruction memory access and buffering:** When fetching multiple instructions per cycle, a variety of complexities are encountered, including the difficulty of fetching multiple instructions require accessing multiple cache lines.

Instruction fetch unit encapsulates this complexity, using prefetch to hide the cost of crossing cache blocks.

Instruction fetch unit provides buffering acting as on-demand unit to provide instructions to issue stage as needed.