Dynamically-Scheduled Machines

• In a **Statically-Scheduled machine**, the compiler schedules all instructions to avoid data-hazards: the ID unit may require that instructions can issue together without hazards, otherwise the ID unit inserts **stalls** until the hazards clear

• This section will deal with **Dynamically-Scheduled machines**, where **hardware-based** techniques are used to detect are remove avoidable data-hazards automatically, to allow '**out-of-order**' execution, and improve performance

• Dynamic scheduling used in the Pentium III and 4, the AMD Athlon, the MIPS R10000, the SUN Ultra-SPARC III; the IBM Power chips, the IBM/Motorola PowerPC, the HP Alpha 21264, the Intel Dual-Core and Quad-Core processors

• In contrast, **static multiple-issue** with compiler-based scheduling is used in the Intel IA-64 **Itanium** architectures

• In 2007, the dual-core and quad-core Intel processors use the Pentium 5 family of **dynamically-scheduled** processors. The Itanium has had a hard time gaining market share.

Dynamic Scheduling - the Idea

(class text - pg 168, 171, 5th ed.)

DIVD	**F0**, F2, F4	
ADDD	F10, **F0**, F8	- data hazard, stall issue for 23 cc
SUBD	F12, F8, F14	- SUBD inherits 23 cc of stalls

• ADDD depends on DIVD; in a static scheduled machine, the **ID unit** detects the hazard and causes the basic pipeline to stall for 23 cc

• The SUBD instruction cannot execute because the pipeline has stalled, even though SUBD does not logically depend upon either previous instruction

• suppose the machine architecture was re-organized to let the SUBD and subsequent instructions "**bypass**" the previous stalled instruction (the ADDD) and proceed with its execution -> we would allow "**out-of-order**" execution

• however, out-of-order execution would allow out-of-order completion, which may allow RW (Read-Write) and WW (Write-Write) data hazards

• a RW and WW hazard occurs when the reads/writes complete in the wrong order, destroying the data. We would need to handle these hazards.

Dynamic Scheduling - the Idea

(class text, pg. 169)

```
DIVD    F0, F2, F4
ADDD    F6, F0, F8
SUBD    F8, F10, F14
MULD    F6, F10, F8
```

• there is a RW data hazard between ADDD and SUBD (ADDD reads **F8**, SUBD writes **F8**, even though there is no logical flow of information between the instructions); if they execute in the wrong order the program is incorrect; in a **statically-scheduled machine**, the **ID Unit** will stall to preserve order of R & W

• there is an WW data-hazard between ADDD and MULD (both write **F6**, even though there is no logical flow of information between the two instructions); if they execute in the wrong order the program is incorrect; in a **statically scheduled machine**, the **ID Unit** will stall to preserve order of W & W

• these problems can be solved, and this code sequence can execute as fast as possible, by using "**Tomasulo's algorithm**" (also called "**Dynamic Scheduling**")

Dynamic Scheduling - the Idea

```
DIVD    F0, F2, F4
ADDD    F6, F0, F8    - unavoidable data hazard, stall 23 cc
SUBD    F8, F10, F14
MULD    F6, F10, F8
```

• Here is the same example ; in a static machine, the **ID unit** will stall the **ADDD** until the hazards clear

• suppose we create some temporary registers, called nF6 and nF8, and we '**rename**' the second occurrences of **F6** and **F8** to use these new registers; this sequence avoids the hazards, and is logically equivalent (gives the same answer):

```
DIVD    F0, F2, F4
ADDD    F6, F0, F8
SUBD    nF8, F10, F14
MULD    nF6, F10, nF8
```

• we can now let the SUBD and MULD instructions execute **out-of-order**, since there are no RW or WW hazards

Tomasulo's Algorithm = Dynamic Scheduling

• Most advanced CPUs (PowerPC, MIPS, Pentium) now implement **Tomasulo's algorithm**, originally proposed in 1960s for large supercomputers

• Basic ideas:

- (i) hardware can unroll loops and execute multiple iterations optimally, without compiler loop unrolling and scheduling

- (ii) hardware automatically bypasses "**data hazards**" without stalling pipelines whenever possible

- (iii) hardware **issues** instructions at maximum rate without data hazard stalls; ID **never stalls due to data-hazards**; it only stalls due to **structural-hazards**

• hardware performs "**register-renaming**" function automatically, eliminating RW and WW data-hazards - this is probably the trickiest part of

Hardware Organization (class text – pg. 173,174)

• How "**dynamic scheduling**" is accomplished:

- (i) several **Functional Units (FUs),** i.e., MULT, DIV, SRT, ADD/SUB, etc

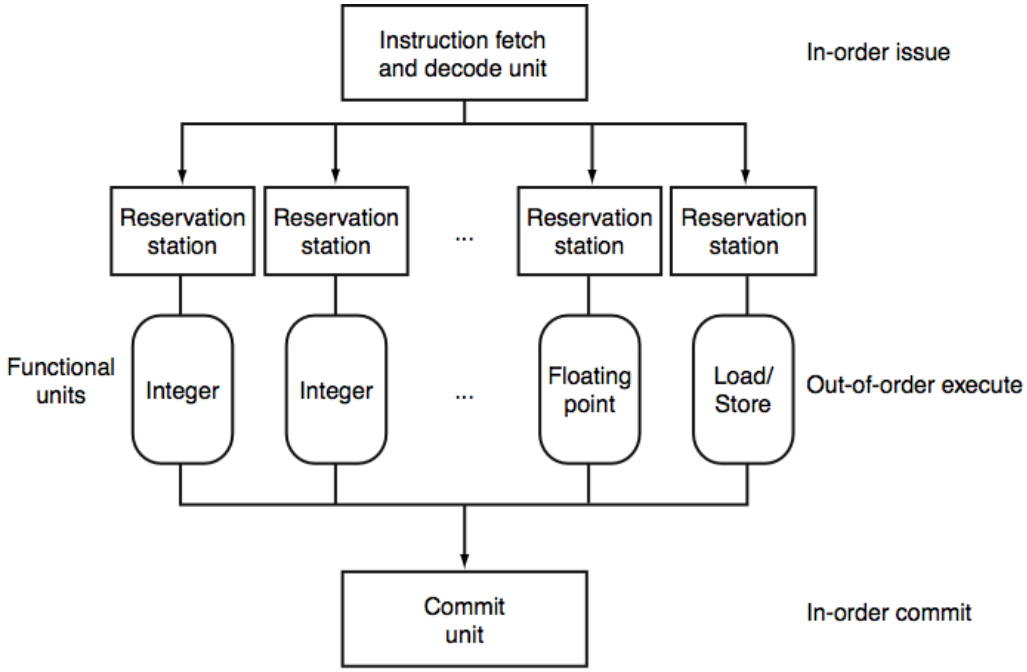
- (ii) each **Functional Unit** contains a "**Reservation Station**", which queues several instructions to be performed and their operands if they are ready. If operands are not ready, it queues pointers to other **FU**s that will **produce** the operands (these pointers are called "**tags**")

-(iii) as an instruction is **issued** at the **Instruction Queue (ID stage)**, its (**operands** are **pre-fetched) OR (tags** are **pre-fetched)** from the registers and stored along with the instruction in the appropriate Reservation Station

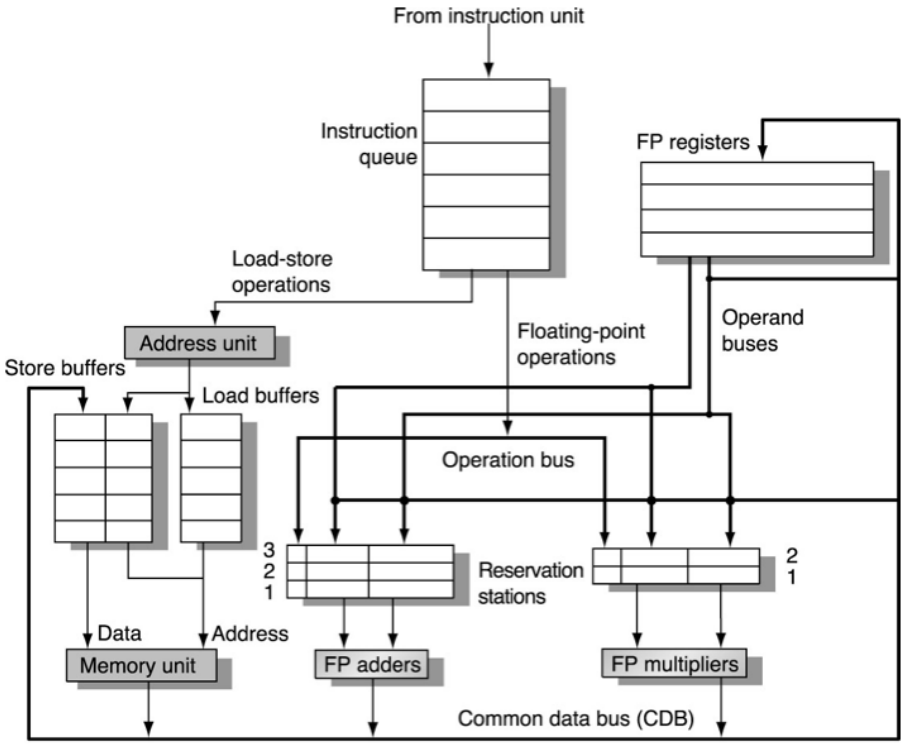
- basically, an instruction copies its operands when it issues, and takes its operands with it to the reservation station, if the operands are ready; if the operand(s) are not ready, the instruction takes an 'I-owe-you' (IOU or "pointer" or 'tag') for the operand and will capture the operand once it is computed using the IOU

- an operand can move directly from a "**producer**" FU to a "**consumer**" RS, bypassing the register file and removing all data-hazards

Hardware Structure Dynamic Scheduling (class text - fig. 6.49, pg 444)



Hardware Structure - More Detail (class text – pg. 173)



Dynamic Scheduling : Tomasulo's Algorithm

- (iv) new results computed in a Functional Unit are **broadcasted** to all Reservation Stations (via a broadcast over the "**Common Data Bus**"), and written back to registers at same time

- Net affect: to eliminate all the RW and WW **data hazards** involved with reading and writing to registers; this solution relies upon broadcasting results as they become available to all Reservation Stations who took an **IOU** for the result

• Tomasulo's algorithm decouples instruction **issue** from instruction **execution**; the algorithm will keep issuing instructions (in the ID stage) as long as there is room in the Reservation Stations; it **never stalls** instruction issues due to data-hazards

• Instruction Issues will only stall if a Reservation Station is full (**structural hazard**)

• A data hazard cannot cause instruction issues to stall in **dynamic-scheduling**; Compare with a **linear static-scheduled pipeline**: If a LOAD results in a cache miss, the entire pipeline must stall until the memory read is completed, which can take 100s cc. The entire pipeline must stall, because the Load cannot move into the Write-Back stage until it is ready to write. Other instructions must stall, because they cannot "**bypass**" the Load.

Three Basic Steps (ref text – pg. 174)

• 3 main steps to complete an instruction, each can take many clock cycles:

• (1) **ISSUE**: get instruction from Instruction Queue; if there is an empty Reservation Station for the instruction (ie ADD, MULT, DIV), then issue the instruction to the RS, and fetch its operands if they are available and forward to the RS; otherwise, forward a "tag" (a pointer to the FU that will generate the operand(s)) to RS (this implements "***register renaming***"). If there is no empty RS for the instruction, we have a **structural-hazard** and must stall pipeline

• (2) **EXECUTE**: at every RS: if operand is not ready, monitor the CDB waiting for that register to be written by the correct producer; copy the result into the RS when it appears on the CBD using the **IOU** (or **tag**) When both operands are ready, execute the instruction if FU is not busy, or label instruction as ready to execute if the FU is busy

• (3) **WRITE RESULT**: at every FU: when a result is computed, write it over the Common Data Bus to the destination register and to any waiting Reservation Stations that have an IOU for that operand

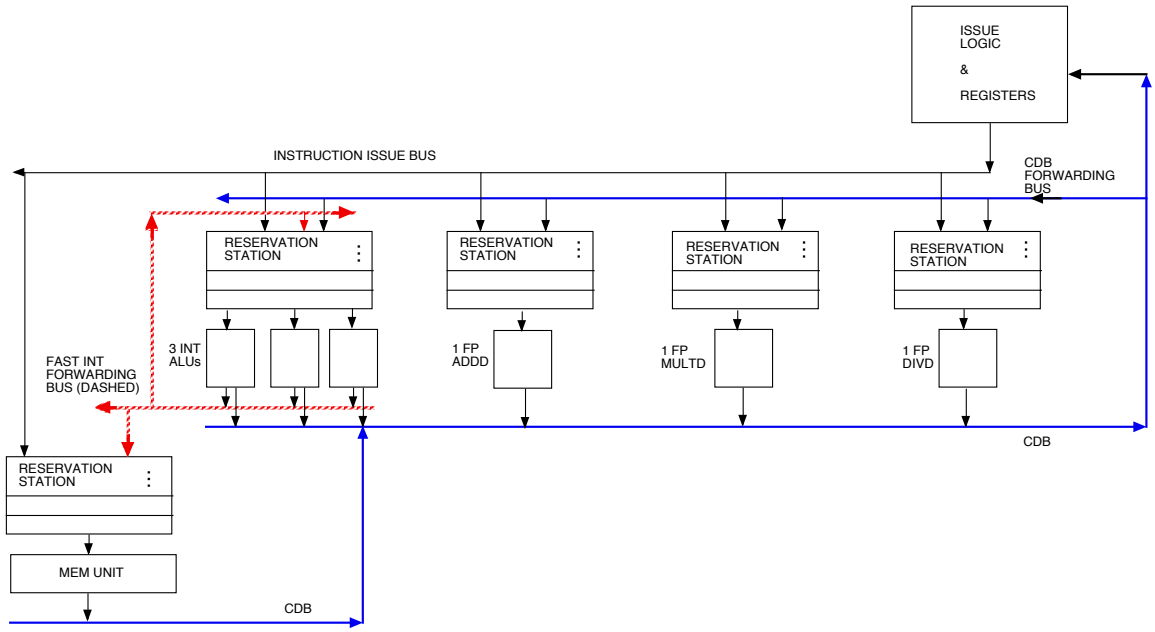
• all WW and RW data-hazards are automatically avoided, through the register renaming process

• data structures to implement the logic are distributed through the Reservation Stations

Comments

- Major advantages of Tomasulo's algorithm:
  - (i) distribution of data-forwarding logic to all reservation stations and CDB
  - (ii) elimination of RW and WW hazards
  -(iii) issues occur at maximum possible rate
  - (iv) execution occurs at maximum possible rate, limited only by unremovable data-hazards or structural-hazards in the original code

Hardware Diagram for Example #1



Example #1- Sequence of Instructions

Single-Issue, Dynamic Scheduling

Assumptions:
(1) FP ADDER is a 4 stage pipeline, FP MULT is an 8 stage pipeline
(2) FP DIV is a 24 stage unit, non-pipelined
(3) Write-Back (WB) takes 1 cc
(3) FP Data forwarding occurs in Write-Back (WB) cc
except for integer forwarding (which takes 0 cc)
(4) the EX column shows the execution times in the integer ALU unit, and the FP units, to save space
(5) Multiple Integer ALU units, to avoid structural hazards
(6) Mem read/write take 1 cc (we have perfect memory with cache hits)
(7) Reservation-stations are infinitely deep (there are no structural hazards)

Instruction	Issue cc	Execute cc	Mem cc	Forwarding/ Write-Back cc	Hazard
LD **F6**, 34(R2)	1	2	3	4	
LD **F2**, 45(R3)	2	3	4	5	
MULTD **F0**, **F2**, F4	3	6..13		14	wait for F2, forwarded in cc 5
SUBD **F8**, **F6**, **F2**	4	6..9		10	wait for F2, forwarded in cc 5
DIVD F10, **F0**, **F6**	5	15..38		39	wait for F0, forwarded in cc 14
ADDD **F6**, **F8**, **F2**	6	11..14		15	wait on F8, forwarded in cc 10

•

Example #2 - Concept of 'Tags' for Operands

EXAMPLE: INITIAL STATE, Registers (and 'IOU's for Operands)

	instruction	Operand A	Operand B
Instruction Queue			
	MULTD F6,F8,F12	F8	F12
j+3	ADDD F6,F8,F12	F8	F12
j+2	MULTD F12,F6,F8	F6	F8
j+1	SUBD F8,F6,F2	F6	F2
j	ADDD F6,F4,F2	F4	F2
ID Unit			

	Register	Valid	Producer	Write-#
F0	0	1		
F2	20	1		
F4	60	1		
F6	0	1		
F8	0	1		
F10	0	1		
F12	0	1		
F14	0	1		

Consider the state of the registers on the left, as the instructions issue. together, a (producer and write-#) form an 'IOU' for an operand

RESERVATION-STATION

FP ADD/SUB PIPELINE

RESERVATION-STATION

FP MULT Pipeline

• state of registers, before any instruction has issued.

Example #2 - Concept of 'Tags' for Operands

EXAMPLE: FINAL STATE, Registers (and 'IOU's for Operands)

	instruction	Operand A	Operand B
Instruction Queue			
	MULTD F6,F8,F12	F8	F12
j+3	ADDD F6,F8,F12	F8	F12
j+2	MULTD F12,F6,F8	F6	F8
j+1	SUBD F8,F6,F2	F6	F2
j	ADDD F6,F4,F2	F4	F2
ID Unit			

	Register	Valid	Producer	Write-#
F0	0	1		
F2	20	1		
F4	60	1		
F6	0	0	FP-MULTD	3
F8	0	0	FP-SUB	1
F10	0	1		
F12	0	0	FP-MULT	1
F14	0	1		

Consider the state of the registers on the left, as the instructions issue.

together, a (producer, a register, and a write-#) form an 'IOU' for an operand

Example: FP-ADD(F6,1) denotes the first write to F6 by the FP-ADD functional unit

RESERVATION-STATION

ADDD F6,F8,F12	FP-ADD(F6,1)	FP-MULT(F12,1)
SUBD F8,F6,F2	FP-ADD(F6,1)	20
ADDD F6,F4,F2	60	20

FP ADD/SUB PIPELINE

RESERVATION-STATION

MULTD F6,F8.F12	FP-SUB(F8,1)	FP-MULT(F12,1)
MULTD F12,F6,F8	FP-ADD(F6,1)	FP-SUB(F8,1)

FP MULT Pipeline

• we assume no functional unit has started execution yet, since we are trying to show the data forwarding using IOUs

Dynamic Loop Unrolling (text, pg 179)

• Consider the loop to multiply a vector by a scalar in F31;

```
Loop:  LD     F0, 0(R1)      ; R1 = pointer to element in memory
       MULD   F4, F0, F31    ; add scalar in F2, store in F4
       SD     0(R1), F4      ; store into memory
       SUBI   R1, R1, #8     ; dec pointer by 8 bytes (size of FP)
       BNEZ   R1, Loop
```

• lets assume "**cancelling branches-PREDICT-TAKEN**" (ie predict that branches are always taken)

• Tomasulo's algorithm will **unroll the loop and issue as many instructions as it can**, until it encounters a **structural-hazard** - effectively decouples the issue from the computation !

• hardware dynamically unrolls the loop and eliminates all RW and WW data-hazards

• Tomasulo's algorithm does not require many temporary FP registers to store intermediate results, since intermediates are **implicitly stored** in the Reservation Station Buffers. Recall that when the compiler unrolled the loop, we needed many extra FP registers to avoid RW and WW hazards.

Example #3 - Automatic Loop Unrolling, Single Issue

(Example done in class, 2012)

Assumptions (same as before)
extra assumptions:
(8) Assume 'Cancelling-Branches, with PREDICT_TAKEN". Assume IF unit updates PC and starts fetching at the top of the loop in the next cc
(9) Assume instructions after a branch cannot start execution until AFTER the branch is resolved (called 'NON-SPECULATIVE' execution)
(10) There are 2 MEM stages, 2 cc deep.
(11) All writebacks and forwardings over the CDB are pipelined and take 2 cc.
(12) Assume CDB has infinite capacity for writebacks

	Instruction	Issue	Execute	Mem	Write-Back	Hazard
loop	LD F0, 0(R1)	1	2	3..4	5..6	
	MULD F4,F0,F2	2	7..14		15..16	wait for F0
1st iteration	SD F4,0(R1)	3	4	17..18		wait for F4
	SUBI R1,R1, #-8	4	5		6..7	
*	BNE R1,loop	5	6			R1 forwarded in 0 cc
	LD F0, 0(R1)	6	7	8...9	10..11	execution starts after branch resolved
	MULD F4,F0,F2	7	12..19		20..21	wait for F0
2nd iteration	SD F4,0(R1)	8	9	22..23		wait for F4
	SUBI R1,R1, #-8	9	10		11..12	
	BNE R1,loop	10	11			
	LD F0, 0(R1)	11	12	13..14	15..16	
	MULD F4,F0,F2	12	17..24		25..26	wait for F0
	SD F4,0(R1)	13	14	27..28		wait for F4
	SUBI R1,T1, #-8	14	15		16..17	
	BNE R1,loop	15	16			

Concept of 'Steady-State' Performance

• Referring to last slide, lets examine loop iterations, to see if a '**steady-state**' is reached

• Here is a definition of '**steady-state**' for a loop: The '**STATE**' of the machine remains constant, when viewed at recurring event times in each loop iteration. The state includes the number of queued instructions in the reservation stations.

• A '**recurrent event time**' is defined as the clock cycle when any event for a recurring instruction starts or ends, ie, the issue clock cycle for the same instruction in every loop, or the start of the execution for the same instruction in every loop

• The '**STATE**' of the machine includes: (a) the number of instructions queued in each type of Reservation Station.

• If a Steady-State is reached, then we can **estimate the machine performance** using this formula:

  • (performance) = (Flops per iteration/clock cycles per iteration) * (Clock Rate)

• This is an estimate, because it ignores the times to **fill-up** the pipelines at the beginning of a loop, and it ignores the time to **flush** the pipelines at the end of a loop

• For large loops, these **fill-up and flush** times become negligible and the performance estimate is quite accurate

Dynamic Loop Unrolling - Comments

• Referring to last slide, lets examine loop iterations, to see if a '**steady-state**' is reached

• The 1st LD starts EX in each loop iteration at times: 2, 7, 12,

• The 1st LD starts MEM in each loop iteration at times: 3, 8, 13,

• The 1st LD starts WB in each loop iteration at times: 5, 10, 15,

• Define '**Delta-T (j, j-1)**' as the difference between the starting clock cycle of 2 identical recurring events in loops (j) and (j-1)

• lets pick an recurring event = cc that the 1st LD starts EX stage in each iteration

• Observe that **Delta-T (2, 1)** = 5 cc

• Observe that **Delta-T (3, 2)** = 5 cc

• Observe that loop 3 appears to be have reached a steady state

• the ending state of the machine for iteration 3 is the same as the ending state of the machine for iteration 2, except all cc's are shifted by +5 cc: The queues in the reservation stations do not grow.

• Lets add another loop iteration to our table and see:

Dynamic Loop Unrolling - Steady-State

Assumptions (same as before)
extra assumptions:
(8) Assume 'Cancelling-Branches, with PREDICT_TAKEN". Assume IF unit updates PC and starts fetching at the top of the loop in the next cc
(9) Assume instructions after a branch cannot start execution until AFTER the branch is resolved (called 'NON-SPECULATIVE' execution)
(10) There are 2 MEM stages, 2 cc deep.
(11) All writebacks and forwardings over the CDB are pipelined and take 2 cc.
(12) Assume CDB has infinite capacity for writebacks

	Instruction	Issue	Execute	Mem	Write-Back	Hazard
loop	LD F0, 0(R1)	1	2	3..4	5..6	
	MULD F4,F0,F2	2	7..14		15..16	wait for F0
1st iteration	SD F4,0(R1)	3	4	17..18		wait for F4
	SUBI R1,R1, #-8	4	5		6..7	
*	BNE R1,loop	5	6			R1 forwarded in 0 cc
	LD F0, 0(R1)	6	**7**	8...9	10..11	execution starts after branch resolved
	MULD F4,F0,F2	7	12..19		20..21	wait for F0
2nd iteration	SD F4,0(R1)	8	9	22..23		wait for F4
	SUBI R1,R1, #-8	9	10		11..12	
	BNE R1,loop	10	11			
	LD F0, 0(R1)	11	**12**	13..14	15..16	execution starts after branch resolved
	MULD F4,F0,F2	12	17..24		25..26	wait for F0
	SD F4,0(R1)	13	14	27..28		wait for F4
	SUBI R1,T1, #-8	14	15		16..17	
	BNE R1,loop	15	16			
	LD F0, 0(R1)	16	**17**	18..19	20..21	execution starts after branch resolved
	MULD F4,F0,F2	17	22..29		30..31	wait for F0
	SD F4,0(R1)	18	19	32..33		wait for F4
	SUBI R1,T1, #-8	19	20		21..22	
	BNE R1,loop	20	21			

• this loop has reached a 'steady-state'. For loop iterations j >= 2, Delta-T = 5 clock cycles.
• the number of instructions queued in each Reservation Station will remain constant, at the same recurring event time within each loop iteration => the queue of instructions in RS does not grow or diminish, but remains constant

4DM4 Example - Dynamic Scheduling - Sequence of Independent MULTIPLIES

- Program : A LOOP of the form X[j] = X[j] * 2

This example illustrates the power of Tomasulo's algorithm, to fetch and issue instructions,
and it also illustrates a long sequence of MULTDs, whose computation is fully pipelined

Assumptions (Data)
- N = loop counter in R3,
- R1 points to the start of vector X in memory
- 2 is a floating point scalar in register F2

Assumptions (Hardware):
- assume a 8 stage MULD unit, fully pipelined
- assume branch-prediction, PREDICT-TAKEN, with cancelling branches; at cc. 6, the branch is predicted taken
- assume NON-SPECULATIVE execution (instructions after a branch must wait for branch resolution before executing)
- assume branches are resolved and forwarded all in 1 cc
- assume a single issue machine
- assume many FUs and deep reservation-stations, so there are no structural hazards
- assume that result forwarding over the CDB Write-Back takes 1 cc

Observations:
- there are 6 instructions per loop iteration
- after 6*N clock cycles, loop instructions must stop issuing
- due to the Cancelling-Branch-Predict-Taken scheme, the machine can start issuing the proper instruction in cc. 7
- there are N MULD instructions, and each is NOT data-dependent on the previous MULD;

You can determine a lot simply by examining the first few loop iterations.

- Q1: At which cc does the loop finish issuing instructions ?
- Q2: At which cc does the loop finish execution (ie all work is done) ?

cc		Instruction		Issue	Execute	Mem	Write-Back	Hazard
1	loop	LD	F0, 0(R1)	1	2	3	4	
2		MULD	F0,F0,F2	2	5..12		13	wait for F0, WB in cc 4
3		SD	F0, 0(R1)	3	4	14		wait for F0, WB in cc 13
4		ADDI	R1,R1, #8	4	5		6	
5		ADDI	R3,R3,#-1	5	6		7	
6		BGEZ	R3, loop	6	8			wait for R3, WB in cc 7
7		LD	F0, 0(R1)	7	9	10	11	wait for branch resolution , before execution
8		MULD	F0,F0,F2	8	12..19		20	wait for F0, WB in cc 11
9		SD	F0, 0(R1)	9	10	21		wait for F0, WB in cc 20
10		ADDI	R1,R1, #8	10	11		12	
11		ADDI	R3,R3,#-1	11	12		13	
12		BGEZ	R3, loop	12	14			wait for R3, WB in cc 13
13		LD	F0, 0(R1)	13	15	16	17	wait for branch resolution , before execution
14		MULD	F0,F0,F2	14	18..25		26	wait for F0, WB in cc 17
15		SD	F0, 0(R1)	15	16	27		wait for F0, WB in cc 26
16		ADDI	R1,R1, #8	16	17		18	
17		ADDI	R3,R3,#-1	17	18		19	
18		BGEZ	R3, loop	18	20			wait for R3, WB in cc 19
19		LD	F0, 0(R1)	19	21	22	23	wait for branch resolution , before execution
20		MULTD	F0,F0,F2	20	24..31		32	wait for F0, WB in cc 23
21		SD	F0, 0(R1)	21	22	33		wait for F0, WB in cc 32
22		ADDI	R1,R1, #8	22				
23		ADDI	R3,R3,#-1	23				
24		BGEZ	R3, loop	24				

Can you see a steady-state pattern emerge in the execution times for each loop iteration, in the steady st
Lets pick one recurring instruction, such as the MULD, and look at its execution times.

- Q3: in iteration j, for j>=2, the MULTD starts at time 6*j cc, and ends 7 cc later
- Q4: in 100-th iteration, MULTD starts execute at cc. 16*100 =600 cc, finishes in cc. 600+7 cc.

- Q5: Can you estimate or compute the number of instructions in the MULTD reservation station, in the steady-state?

4DM4 Example - Dynamic Scheduling - Sequence of DEPENDENT MULTIPLIES

- Program : A LOOP of the form X[j] = X[j] * X[j-1]*X[j-2]*.....*X[1]

This example illustrates the power of Tomasulo's algorithm, to fetch and issue instructions,
and it also illustrates a long sequence of MULTDs, whose computation is fully pipelined

Assumptions (Data)
- N = loop counter in R3,
- R1 points to the start of vector X in memory

Assumptions (Hardware):
- assume a 8 stage MULD unit, fully pipelined
- assume branch-prediction, PREDICT-TAKEN, with cancelling branches; at cc. 6, the branch is predicted taken
- assume NON-SPECULATIVE execution (instructions after a branch must wait for branch resolution before executing)
- assume branches are resolved and forwarded all in 1 cc
- assume a single issue machine
- assume many FUs and deep reservation-stations, so there are no structural hazards
- assume that result forwarding over the CDB Write-Back takes 1 cc

Observations:
- there are 6 instructions per loop iteration
- after 6*N clock cycles, loop instructions must stop issuing
- due to the Cancelling-Branch-Predict-Taken scheme, the machine can start issuing the proper instruction in cc. 7
- there are N MULD instructions, and each is NOT data-dependent on the previous MULD;

You can determine a lot simply by examining the first few loop iterations.

- Q1: At which cc does the loop finish issuing instructions ?
- Q2: At which cc does the loop finish execution (ie all work is done) ?

cc		Instruction	Issue	Execute	Mem	Write-Back	Hazard
1	loop	LD F2, 0(R1)	1	2	3	4	
2		MULD F0,F0,F2	2	5..12		13	wait for F0, WB in cc 4
3		SD F0, 0(R1)	3	4	14		wait for F0, WB in cc 13
4		ADDI R1,R1, #8	4	5		6	
5		ADDI R3,R3,#-1	5	6		7	
6		BGEZ R3, loop	6	8			wait for R3, WB in cc 7
7		LD F2, 0(R1)	7	9	10	11	wait for branch resolution, before execution
8		MULD F0,F0,F2	8	14..21		22	wait for F0, WB in cc 13
9		SD F0, 0(R1)	9	10	23		wait for F0, WB in cc 22
10		ADDI R1,R1, #8	10	11		12	
11		ADDI R3,R3,#-1	11	12		13	
12		BGEZ R3, loop	12	14			wait for R3, WB in cc 13
13		LD F2, 0(R1)	13	15	16	17	wait for branch resolution, before execution
14		MULD F0,F0,F2	14	23..30		31	wait for F0, WB in cc 22
15		SD F0, 0(R1)	15	16	32		wait for F0, WB in cc 31
16		ADDI R1,R1, #8	16	17		18	
17		ADDI R3,R3,#-1	17	18		19	
18		BGEZ R3, loop	18	20			wait for R3, WB in cc 19
19		LD F2, 0(R1)	19	21	22	23	wait for branch resolution, before execution
20		MULTD F0,F0,F2	20	32... 39		40	wait for F0, WB in cc 31
21		SD F0, 0(R1)	21	22	41		wait for F0, WB in cc 40
22		ADDI R1,R1, #8	22				
23		ADDI R3,R3,#-1	23				
24		BGEZ R3, loop	24				

Can you see a steady-state pattern emerge in the execution times for each loop iteration, in the steady st
Lets pick one recurring instruction, such as the MULD, and look at its execution times.

- Q3: in iteration j, for j>=2, the MULTD starts at time 5+(j-1)*9 cc, = 9*j-4 cc, and ends 7 cc la
- Q4: in 100-th iteration, MULTD starts execute at cc. 100*9-4= 896 cc, finishes in cc. 896+7 cc.

- Q5: Can you estimate or compute the number of instructions in the MULTD reservation station, in the steady-state?

Observations - Loop with Unavoidable Data-Hazards

• The MULTDs in the last loop have **unavoidable data-hazards**. They cannot be removed. Each MULTD must wait the result of the previous MULTD.

• Observe that for j >= 2, **Delta-T (j, j-1) = 6 cc** for all instructions **except** for the MULTD

• Lets define our recurring event as the time the MULTD starts Execution

• Observe that for j>=2, **Delta-T (j, j-1) = 9 cc** for the MULTD EX

• The MULTDs are executing at a much slower rate than all the other instructions, due to their unavoidable data-dependencies

• If we run a loop with 1000 iterations, the issues take 6*1000 cc. The last integer instructions will finish execution at time roughly 6*1000 +1 cc and will disappear.

• At time = 6000 cc, only floor(6000/9cc) = 667 MULTDs will have finished, and there will be roughly 333 MULTDs left in the MULTD Reservation stations, and 333 SDs in the MEM reservation stations awaiting execution (assuming RSs have infinite capacity)

• At this point, all the instructions have issued, the processor has moved on to a new loop, and the MULTDs and SDs are left to finish as soon as they possibly can

Final Thoughts

• Tomasulo's algorithm allows **out-of-order** execution; instructions issue as soon a possible, instructions execute as soon as possible

• Avoidable data-hazards are automatically removed. **Unavoidable Data-hazards** are still present ; suppose we have a long sequence of **data-dependent** instructions: In a **static-scheduled pipeline**, the ID Unit **stalls the issue** of each instruction until the data hazard clears. With **dynamically-scheduled pipeline**, all instructions issue as soon as possible, and the execution proceeds **as soon as possible**

• The main advantage of Tomasulo's algorithm is that it allows **out-of-order execution** : it lets **other** instructions without data-dependencies execute **out-of-order** when there are data-hazards, and all executions happen as soon as they possibly can

• Good compiler scheduling can approach the performance of Tomasulo's algorithm.

• Tomasulo's algorithm is expensive in hardware. Nevertheless, most supercomputers, the Motorola PowerPC, the Pentiums and the Intel dual-core and quad-core processors (dual Pentiums) all rely on Tomasulo's algorithm for performance.

• In contrast, the 'next generation' processor from Intel/HP, the **Itanium** processor, has moved to a **static-scheduled pipeline**, to avoid hardware cost of Tomasulo's algorithm. However, the commercial acceptance of the Itanium has 'stalled', and time will tell which technique (static vs dynamic scheduling) will rule.

Dynamic Scheduling with Multiple-Issue

- machines which use Tomasulo's algorithm can easily be modified to support multiple-issue
- the next example illustrates a multiple-ISSUE machine
- a problem that arises with multiple issue is branch hazards - branches arrive much faster due to multiple issue; we might have 1 branch per cc
- many loops have a branch at the bottom of the loop, back to the top of the loop
- to keep issuing at maximum rate, we will use BRANCH-PREDICTION, ie predict the branch as taken, so we start the issue for the next loop iteration even before we resolve the branch
- however, we avoid executing these instructions until we resolve the branch
- we call this mode of execution 'NON-SPECULATIVE EXECUTION'

Example - Non-Speculative Execution

Loop Unrolling, 5-Issue Dynamic Scheduling

Assumptions

* MULTD unit is a 6 stage pipeline
* Assume branch-prediction, with PREDICT_TAKEN
* Assume NON-SPECULATIVE Execution
  (ie instructions after a branch cannot start execution until AFTER the branch is resolved)
* There are 2 MEM units (2-stage pipeline each)
* 4 INT ALUs; depth INT Reservation-Station = 8 instructions
* INT forwarding takes 0 cc, WB takes 2 cc

	Instruction	Issue	Execute	Mem	Write-Back	Hazard
loop	LD F0, 0(R1)	1	2	3..4	5..6	
	MULD F4,F0,F2	1	7..12		13..14	wait for F0, ready end cc 6
1st iteration	SD F4,0(R1)	1	2	15..16		wait for F4, ready end cc 14
	SUBI R1,R1, #-8	1	2		3..4	
*	BNE R1,loop	1	**3**			wait for R1, ready end cc 2
	LD F0, 0(R1)	2	4	5..6	7..8	execution starts after branch resolved in cc 3
	MULD F4,F0,F2	2	9..14		15..16	wait for F0, ready at end of cc 8
2nd iteration	SD F4,0(R1)	2	4	17..18		wait for F4, ready end of cc 16
	SUBI R1,R1, #-8	2	4		5..6	
	BNE R1,loop	2	**5**			wait for R1, ready end 4
						at end cc 2, depth INT-RS = 5 inst.ructions
	LD F0, 0(R1)	3	6	7..8	9..10	execution starts after branch resolved in cc 5
	MULD F4,F0,F2	3	11..16		17..18	wait for F0, ready at end of cc 10
	SD F4,0(R1)	3	6	19..20		wait for F4, ready at end of cc 18
	SUBI R1,R1, #-8	3	6		7..8	
	BNE R1,loop	3	**7**			**at end of cc 3, depth INT-RS = 8 instructions**

LD	F0, 0(R1)	3	6	7..8	9..10	execution starts after branch resolved in cc 5
MULD	F4,F0,F2	3	11..16		17..18	wait for F0, ready at end of cc 10
SD	F4,0(R1)	3	6	19..20		wait for F4, ready at end of cc 18
SUBI	R1,R1, #-8	3	6		7..8	
BNE	R1,loop	3	7			**at end of cc 3, depth INT-RS = 8 instructions**
LD	F0, 0(R1)	4	8	9..10	11..12	execution starts after branch resolved in cc 7
MULD	F4,F0,F2	4	13..18		19..20	wait for F0, ready at end of cc 12
SD	F4,0(R1)	4	8	21..22		wait for F4, ready at end of cc 20
SUBI	R1,R1, #-8	4	8		9..10	
BNE	R1,loop	5	9			**Struct hazard - depth INT-RS = 8 instructions; stall other INT instructions, until INT-RS has room**
LD	F0, 0(R1)	6	10	11..12	13..14	execution starts after branch resolved in cc 9
MULD	F4,F0,F2	6	15..20		21..22	wait for F0, ready at end of cc 14
SD	F4,0(R1)	6	10	23..24		wait for F4, ready at end of cc 22
SUBI	R1,R1, #-8	6	10		11..12	
BNE	R1,loop	7	11			**Struct hazard - depth INT-RS = 8 instructions;**
This loop DOES reach a STEADY-STATE: Issues are happening with same Delta-T as execution. DELTA-T = 2 cc						
The # instructions queued up in RS reaches a steady-state = 8 for cc 5, 7, 9, 11, etc						
Performance:		(1 FLOP)/(2 cc) * 4 GHz = 2.0 GFLOP/sec				

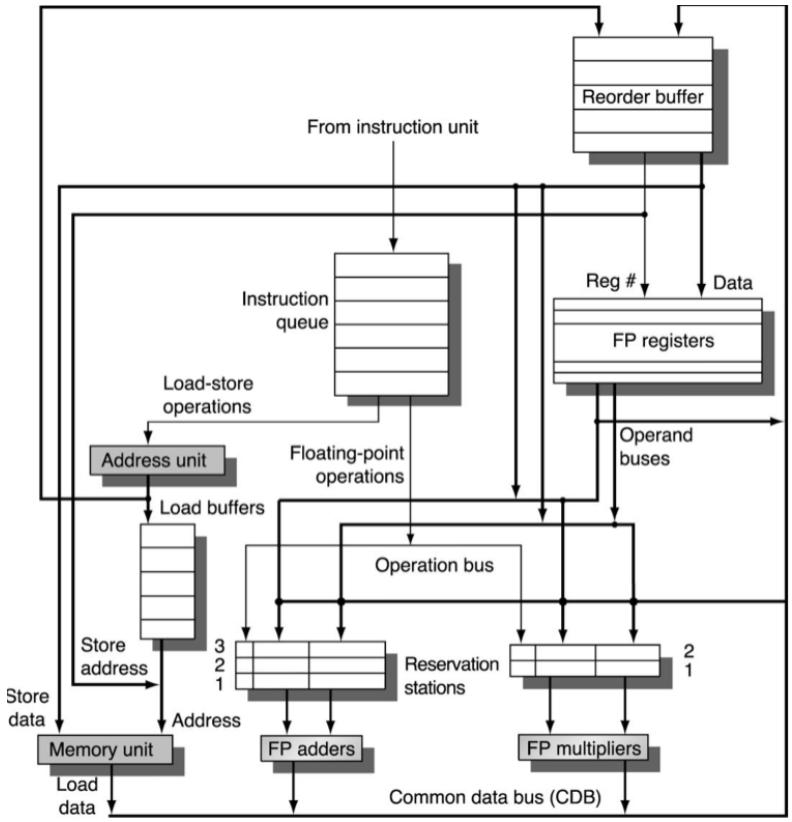
- observe that there are structural hazards on the INT Reservation-Station at clock cycles 5, 7, 9, 11, etc
- INT instructions fill up in the RS and cannot execute until the prior branches resolve
- the loop appears to reach a steady state at clock cycles 4,5, then 6,7, etc
- it takes 2 ccs to issue 5 instructions, due to structural hazards

Hardware-Based 'Speculative Execution' (class text – pg. 183)

- recall branches result in large slowdowns for real machines
- in a wide n-issue machine, there may be a branch in every clock cycle
- to exploit more parallelism, we need to remove branch hazards
- solution: use "**SPECULATIVE EXECUTION**" on outcome of branch; predict the branch outcome, and start execution according to your prediction
- similar to the "**Branch-Prediction**" concept we looked at in the regular linear pipeline
- The main difference is that we now allow instructions to execute and do a '**tentative write-back'** before we know the outcome of the branch; call these results '**speculative results'**
- other instructions might use these **speculative results** and start execution, so they also become speculative instructions
- **If** our branch prediction was wrong, we must have special hardware support to "remove" all speculated results
- **Speculation** is used in latest PowerPC chip, MIPS chip, Intel Pentium 3 and Pentium 4 chips, the DEC Alpha 21264, and the AMD Athlon chips
- Basic idea: hardware must keep track of "**speculated instructions**"; once the branch outcome is resolved, the speculated instructions dependent on that branch can be either (1) kept if the branch prediction was correct or (b) be completely undone, with all registers restored to their proper values
- If we keep the results, we say the speculated instructions go through the "**Commit**" stage

Hardware-Based Speculation (class text - pp. 183-187)

- add a new hardware block called the "**Re-Order** Buffer"
- when a speculated instructions finishes execution, it writes it results to the **Re-Order** buffer
- the Re-Order buffer stores all speculated instruction results until we know if they are correct or not
- when the branch outcome is determined, if the prediction was correct then the speculated instructions "**Commit**", and remove their results from the **Re-Order buffer** and copy the results to the Registers
- If the branch was predicted incorrectly, the results of the speculated instructions, which are held in the Re-Order buffer, can be erased since they are useless
- We add the constraint that speculated instructions must commit "**in-order**" to preserve the correctness of the program
- (pg 227 text) entries in the re-order buffer have 4 fields; the speculated instruction, the destination to be written (registers for ALU operations, memory for STORE instructions), the value to be written back, and the result-ready bit (= '1' when the instruction has completed execution and the value field is valid)



MIPS machine with Tomasulo's Alg and Speculation,

(class text – pg. 185)

Main change with speculation is the addition of the **Re-order buffer** (ROB)

Example - Speculative Execution

Loop Unrolling, 5-Issue Dynamic Scheduling

Assumptions (mostly the same as before)

* MULTD unit is a 6 stage pipeline
* Assume Branch-Prediction with PREDICT_TAKEN
* Assume SPECULATIVE Execution
  (ie instructions after a branch start execution asap)
* There are 2 MEM units (2 pipeline stages each)
* 4 INT ALUs; depth INT Reservation-Station = 8 instructions
* INT forwarding takes 0 cc, WB takes 2 cc

	Instruction	Issue	Execute	Mem	Write-Back	Commit flag	1cc COMMIT	Hazard
loop	LD F0, 0(R1)	1	2	3..4	5..6			
	MULD F4,F0,F2	1	7..12		13..14			wait for F0, ready end cc 6
1st iteration	SD F4,0(R1)	1	2	15..16				wait for F4, ready end cc 14
	SUBI R1,R1, #-8	1	2		3..4			
*	BNE R1,loop	1	**3**					wait for R1, ready end cc 2
	LD F0, 0(R1)	2	3	4..5	6..7	4		execution starts after branch resolved in cc 3
	MULD F4,F0,F2	2	8..13		14..15	4		wait for F0, ready at end of cc 7
2nd iteration	SD F4,0(R1)	2	3	16..17		4		wait for F4, ready end of cc 15
	SUBI R1,R1, #-8	2	3		4..5	4		
	BNE R1,loop	2	**4**			4		wait for R1, ready end 3
								Depth INT-RS = 5 instructions
	LD F0, 0(R1)	3	4	5..6	7..8	5		execution starts after branch resolved in cc 4
	MULD F4,F0,F2	3	9..14		15..16	5		wait for F0, ready at end of cc 8
	SD F4,0(R1)	3	4	17..18		5		wait for F4, ready at end of cc 16
	SUBI R1,R1, #-8	3	4		5..6	5		
	BNE R1,loop	3	**5**			5		wait for R1, ready end 4
								DEPTH INT-RS = 5 instructions

Speculative machine

LD F0, 0(R1)	3	4	5..6	7..8	5		execution starts after branch resolved in cc 4
MULD F4,F0,F2	3	9..14		15..16	5		wait for F0, ready at end of cc 8
SD F4,0(R1)	3	4	17..18		5		wait for F4, ready at end of cc 16
SUBI R1,R1, #-8	3	4		5..6	5		
BNE R1,loop	3	**5**			5		wait for R1, ready end 4
							DEPTH INT-RS = 5 instructions
LD F0, 0(R1)	4	5	6..7	8..9	6		execution starts after branch resolved in cc 5
MULD F4,F0,F2	4	10..15		16..17	6		wait for F0, ready at end of cc 9
SD F4,0(R1)	4	5	18..19		6		wait for F4, ready at end of cc 17
SUBI R1,R1, #-8	4	5		6..7	6		
BNE R1,loop	4	6			6		wait for R1, ready end 5
							DEPTH INT-RS = 5 instructions
LD F0, 0(R1)	5	6	7..8	9..10	7		execution starts after branch resolved in cc 6
MULD F4,F0,F2	5	11..16		17..18	7		wait for F0, ready at end of cc 16
SD F4,0(R1)	5	6	19..20		7		wait for F4, ready at end of cc 25
SUBI R1,R1, #-8	5	6		7..8	7		
BNE R1,loop	5	7			7		wait for R1, ready end 6
							DEPTH INT-RS = 5 instructions

This loop DOES reach a STEADY-STATE: Issues are happening with same Delta-T as execution. DELTA-T = 3 cc
The # instructions queued up in RS is increasing, and must fill up and cause Structural Hazards at some point.

Performance: (1 FLOP)/(3 cc) * 4 GHz = 1.33 GFLOP/sec

Note: delta-t = 1 cc in above

- observe that the structural hazards on the INT Reservation-Station have been removed by speculation, since the INT instructions can execute without waiting for branches to resolve, thereby creating more space in the INT-RS
- the machine now issues at the optimal rate (5 instructions per cc), and executes at the optimal rate

Notes

Real Stuff: The Pentium 4 Architecture

(Comp Org. text - section 6.10, class text - pg 259, 3rd ed.)

- The Pentium 4 is a dynamically scheduled speculative pipelined machine that executes the old Intel IA-32 instruction set.
- It translates each "difficult-to-pipeline" IA-32 instruction into a series of pipelinable RISC "micro-ops"
- Up to 3 IA-32 instructions are fetched, decoded and translated to micro-ops per clock cycle
- if an IA-32 instruction requires more than 4 micro-ops, its translation takes multiple clock cycles
- Micro-ops are dynamically scheduled using Tomasulo's algorithm, and therefore execute out-of-order
- The issue stage can issue up to 3 micro-ops per clock cycle; the commit stage can complete up to 3 micro-ops per second
- The P4 gains its advantage over the P3 as follows:
- deeper pipeline: 20 pipeline stages for the P4, versus 10 stages for the P3
- more functional units (7 units for the P4, versus 5 units for the P3)
- use of a trace cache, to speed up micro-op translation

The Pentium 4 Architecture

• The P4 architecture uses speculation and therefore requires a re-order buffer.

• The P4 uses register-renaming, which is inherent in Tomasulo's algorithm.

• The IA-32 instruction set only has 8 general purpose architectural registers, so running out of registers is a problem. The P4 uses 128 general purpose registers

• P4 allows up to 126 microps to be outstanding at any time, including 48 loads and 24 stores

• For comparison, the IBM PowerPC chip allows up to 400 instructions to be outstanding at any time

• The FP unit in Fig 6.50 (2 slides ahead) actually consists of 2 functional units, so there are 7 functions units in total

• The FP Unit handles the special MMX and SSE2 instruction set additions

• There are 2 Integer ALU units, which operate at twice the processor clock rate, so that 4 integer ops can be computed per clock tick

The Pentium 4 Architecture (ref text - pg 268, 3rd ed.)

• The P4 architecture requires 2 clock cycles just to drive results across the chip over the busses

• The P4 ALU and data cache operate at twice the clock rate, to lower latency; this high-speed operation is essential to lower potential stalls due to very deep pipeline

• The P4 has a **Branch-Target-Buffer (BTB)*** in the IF unit that is 8 times larger than the P3 BTB, to lower branch mis-prediction rates. It also uses an improved prediction algorithm.

• In 2002: the P4 had a smaller L1 data cache and larger L2 cache with higher memory bandwidth (compared to Pentium 3), which should offset the smaller L1 cache

• The P4 implements the new Intel "SSE2" floating point instructions that allow 2 FP operations to be done per clock tick. This boosts FP performance considerably.

• (We will discuss BTBs in a future lecture on Branches)

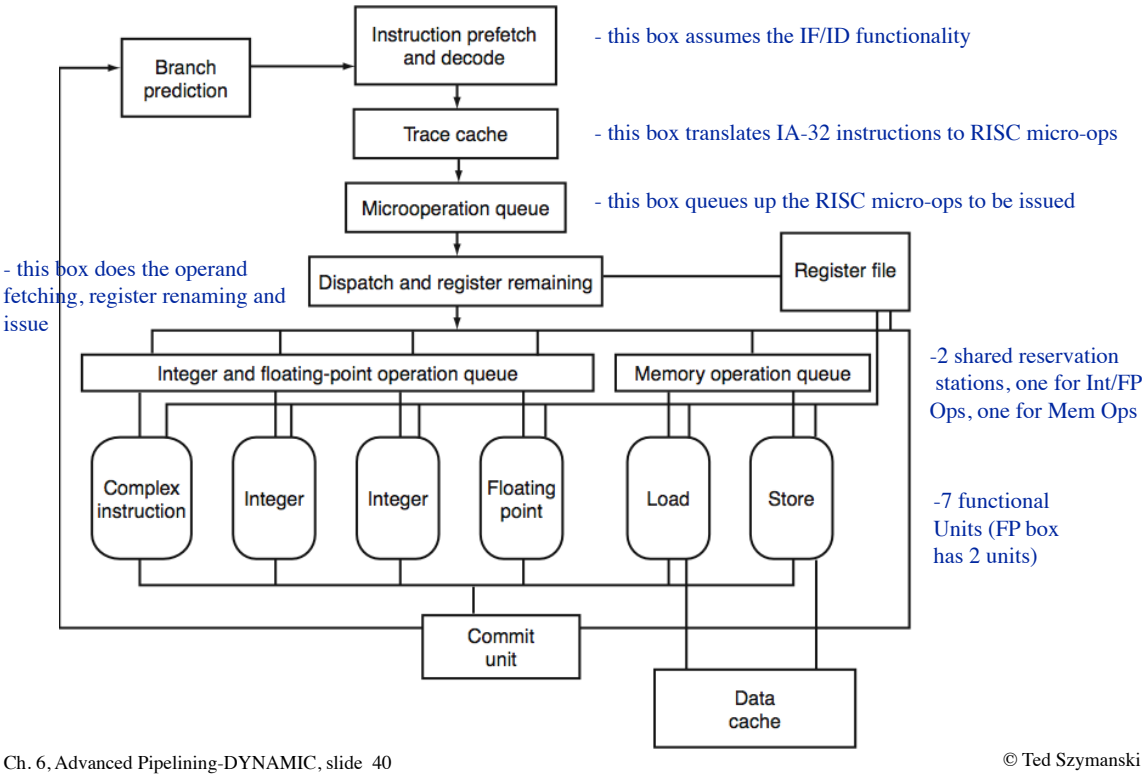
Intel Pentium Processors (vlass text - fig. 3.47, pg 260, 3rd ed.)

Processor	First ship date	Clock rate range	L1 cache	L2 cache
Pentium Pro	1995	100–200 MHz	8 KB instr. + 8 KB data	256 KB–1024 KB
Pentium II	1998	233–450 MHz	16 KB instr. + 16 KB data	256 KB–512 KB
Pentium II Xeon	1999	400–450 MHz	16 KB instr. + 16 KB data	512 KB–2 MB
Celeron	1999	500–900 MHz	16 KB instr. + 16 KB data	128 KB
Pentium III	1999	450–1100 MHz	16 KB instr. + 16 KB data	256 KB–512 KB
Pentium III Xeon	2000	700–900 MHz	16 KB instr. + 16 KB data	1 MB–2 MB

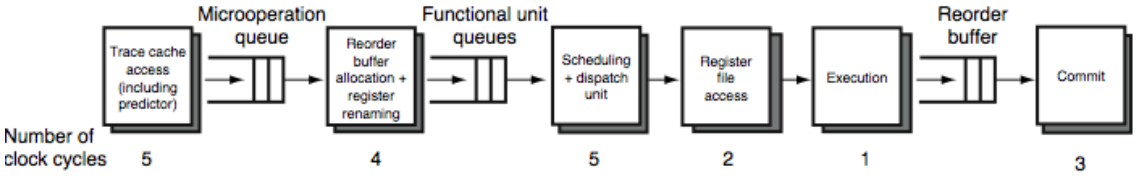
Figure 3.47 The Intel processors based on the P6 microarchitecture and their important differences. In the Pentium Pro, the processor and specialized cache SRAMs were integrated into a multichip module. In the Pentium II, standard SRAMs are used. In the Pentium III, there is either an on-chip 256 KB L2 cache or an off-chip 512 KB cache. The Xeon versions are intended for server applications; they use an off-chip L2 and support multiprocessing. The Pentium II added the MMX instructions extension, while the Pentium III add the SSE extension.

• The Pentium 3 has both an L1 cache and an L2 cache. The L1 cache is about 32 Kbytes, and holds both instructions and data together in one cache. The L2 cache is 'off-chip' in some of the processors. The L2 cache is considerably larger than the L2 cache, usually around 1 Mbyte in size.

Pentium 4 Architecture (Comp. Org. text - fig. 6.50, pg 449)



Pentium 4 Pipeline (Comp. Org. text - fig. 6.51, pg 449)



- this figure shows the typical flow through the P4 for a typical integer instruction. The average number of clock cycles spent in each major unit are shown

- average INT instruction requires 20 cc to complete, corresponding to a 20 stage pipeline

- the major buffers where instructions wait are also shown

Pentium 4 Performance (class text - pg 262, 3rd ed.)

• Several factors improve the performance of P4 vs P3:

• 20 stage pipeline, aggressive multiple issue, dynamic scheduling, speculation

• low latencies for back-to-back operations (0 cc for ALU ops, 2 cc for LOADs)

• Several factors limit the performance of the P4 :

• Often less than 3 32-bit instructions can be fetched, due to **cache misses**

• Often less than 3 32-bit instructions can "issue" per clock cycle, since one of the 3 instructions generated more micro-ops than it is allowed to (there is fast micro-op generation for most instructions, but not all)

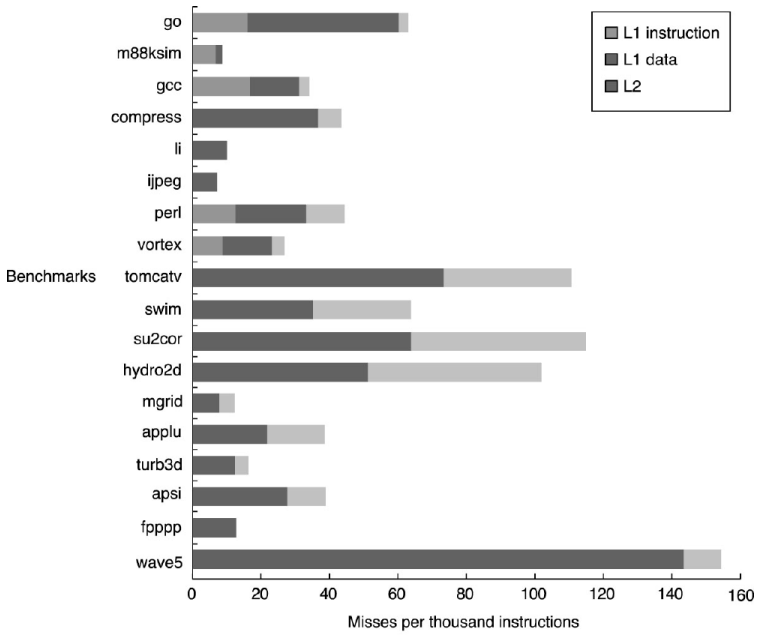
• Not all the micro-ops can issue in a clock cycle due to **structural hazards**, ie a shortage of reservation stations or a shortage of re-order buffers

• Unavoidable **Data dependencies** can lead to stalls

• Data cache misses can lead to stalls, (ie all instructions queued in reservation stations can be stalled waiting for this data and none will execute)

• **Branch Mispredictions** cause stalls, since the pipeline must be flushed and refilled.

Cache Misses (class text - fig. 3.53, pg 266, 3rd ed.)



• on average, L1 and L2 cache miss rates are usually between 10 and 50 misses per 1,000 instructions, a miss rate of 1% - 5%. The L2 cache miss rate is smaller than L1 miss rate because it is larger, but L2 cache misses incur considerably larger stalls, about 5 times as large as an L1 miss

Notes