4

High Performance Processor Design Techniques

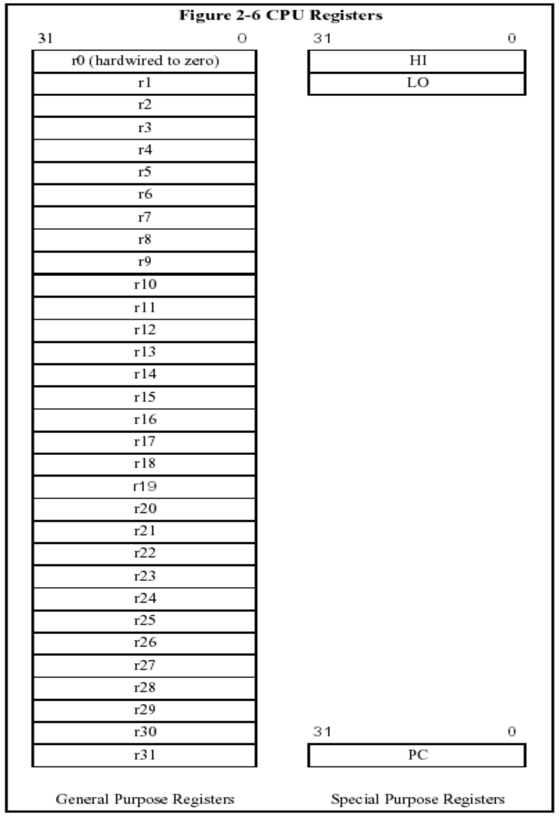
Pipeline Design Technique

- Some examples
  - MIPS32 Architecture for Programmers
    - Volume 1: Introduction to the MIPS Architecture
    - Volume 2: Instruction Set
    - Volume 3: Privileged Resource Architecture
  - MIPS32 4K Processor Core Family – Software User's Manual
  - MIPS32 4Kc Datasheets

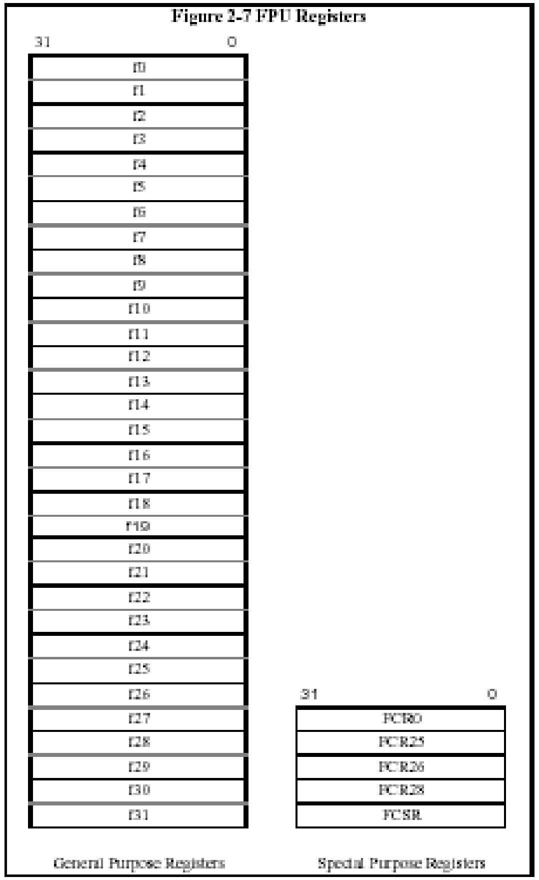
The MIPS Processor Core

- Available as:
  - Synthesizable core
  - Hardcore
  - Standard products
- A RISC architecture
  - A simple load/store instruction set
    - Design for pipelining efficiency
    - An easily decoded instruction set
- Implementation via
  - Pipeline
  - Super-pipeline
  - VLIW pipeline

- Defines the following CPU registers:
  - 32 32-bit general purpose registers (GPRs)
  - a pair of special-purpose registers to hold the results of integer multiply, divide, and multiply-accumulate operations (HI and LO)
  - a special-purpose program counter (PC), which is affected only indirectly by certain instructions - it is not an architecturally-visible register.



MIPS Architecture



- The MIPS32 Architecture defines the following FPU registers:
  - 32 32-bit floating point registers (FPRs). All 32 registers are available for use in single-precision floating point operations. Double-precision floating point values are stored in even-odd pairs of FPRs.
  - Five FPU control registers are used to identify and control the FPU.

ISA

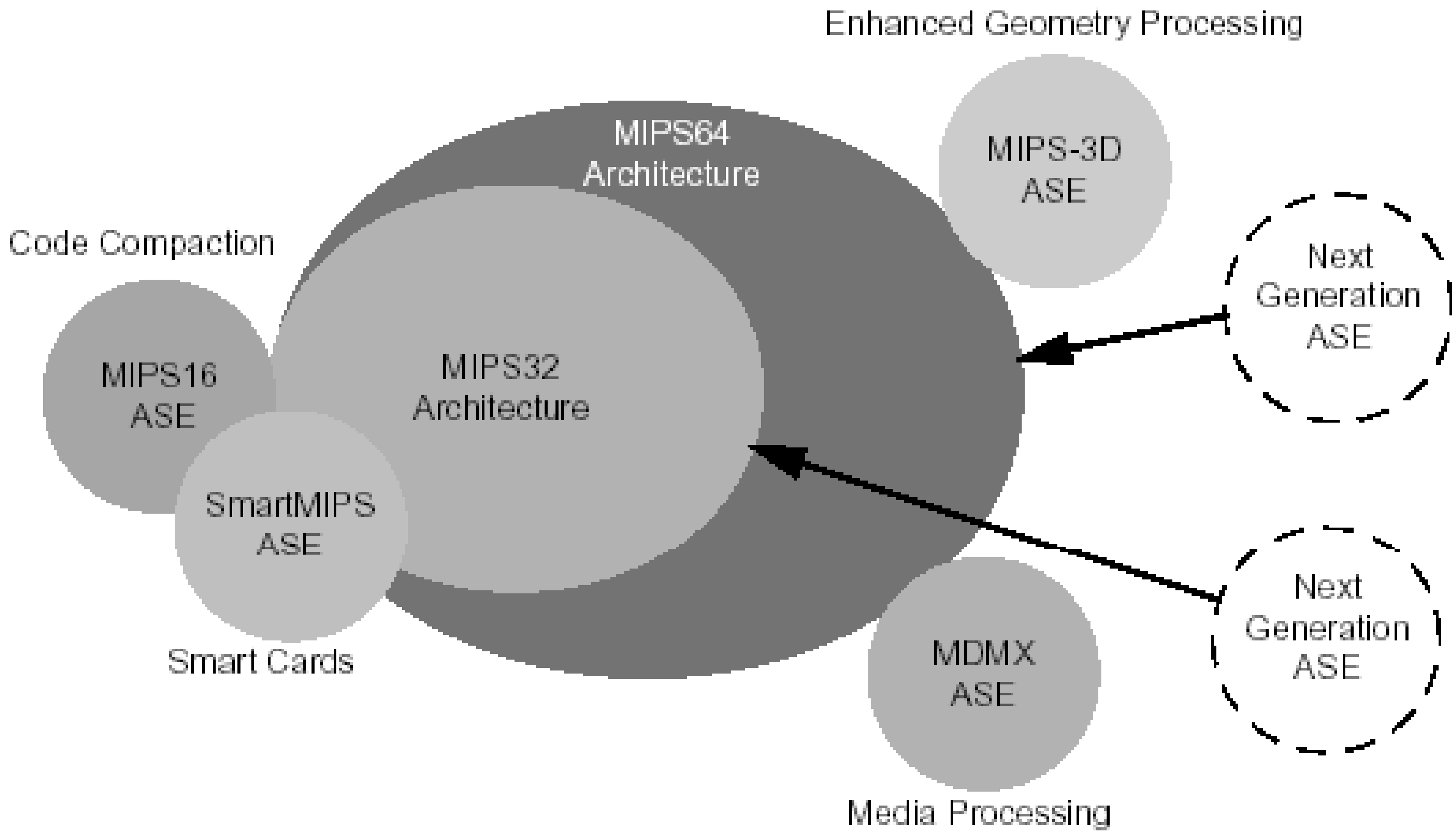


Figure 3-1 MIPS ISAs and ASEs

CPU Instructions, Grouped By Function

- CPU instructions are organized into the following functional groups:
  - Load and store
  - Computational
  - Jump and branch
  - Miscellaneous
  - Coprocessor
- Each instruction is 32 bits long.

Load and Store Instructions

Mnemonic	Instruction
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
SB	Store Byte
SH	Store Halfword
SW	Store Word

Aligned CPU load and store instructions

Mnemonic	Instruction
LWL	Load Word Left
LWR	Load Word Right
SWL	Store Word Left
SWR	Store Word Right

Unaligned CPU load and store instructions

Mnemonic	Instruction
LL	Load Linked Word
SC	Store Conditional Word

Atomic update CPU load and store instructions

Mnemonic	Instruction
LDCz	Load Doubleword to Coprocessor-z, z = 1 or 2
LWCz	Load Word to Coprocessor-z, z = 1 or 2
SDCz	Store Doubleword from Coprocessor-z, z = 1 or 2
SWCz	Store Word from Coprocessor-z, z = 1 or 2

Co-processor load and store instructions

CPU Arithmetic Instructions

Mnemonic	Instruction
ADD	Add Word
ADDI	Add Immediate Word
ADDIU	Add Immediate Unsigned Word
ADDU	Add Unsigned Word
CLO	Count Leading Ones in Word
CLZ	Count Leading Zeros in Word
DIV	Divide Word
DIVU	Divide Unsigned Word
MADD	Multiply and Add Word to Hi, Lo
MADDU	Multiply and Add Unsigned Word to Hi, Lo
MSUB	Multiply and Subtract Word to Hi, Lo
MSUBU	Multiply and Subtract Unsigned Word to Hi, Lo
MUL	Multiply Word to GPR
MULT	Multiply Word
MULTU	Multiply Unsigned Word
SLT	Set on Less Than
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
SLTU	Set on Less Than Unsigned
SUB	Subtract Word
SUBU	Subtract Unsigned Word

CPU Instructions

Table 3-8 CPU Trap Instructions

Mnemonic	Instruction
BREAK	Breakpoint
SYSCALL	System Call
TEQ	Trap if Equal
TEQI	Trap if Equal Immediate
TGE	Trap if Greater or Equal
TGEI	Trap if Greater of Equal Immediate
TGEIU	Trap if Greater or Equal Immediate Unsigned
TGEU	Trap if Greater or Equal Unsigned
TLT	Trap if Less Than
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal
TNEI	Trap if Not Equal Immediate

Table 3-2 CPU Branch and Jump Instructions

Mnemonic	Instruction
B	Unconditional Branch
BAL	Branch and Link
BEQ	Branch on Equal
BGEZ	Branch on Greater Than or Equal to Zero
BGEZAL	Branch on Greater Than or Equal to Zero and Link
BGTZ	Branch on Greater Than Zero
BLEZ	Branch on Less Than or Equal to Zero
BLTZ	Branch on Less Than Zero

Table 3-2 CPU Branch and Jump Instructions

Mnemonic	Instruction
BLTZAL	Branch on Less Than Zero and Link
BNE	Branch on Not Equal
J	Jump
JAL	Jump and Link
JALR	Jump and Link Register
JR	Jump Register

Table 3-3 CPU Instruction Control Instructions

Mnemonic	Instruction
NOP	No Operation
SSNOP	Superscalar No Operation

Table 3-7 CPU Shift Instructions

Mnemonic	Instruction
SLL	Shift Word Left Logical
SLLV	Shift Word Left Logical Variable
SRA	Shift Word Right Arithmetic
SRAV	Shift Word Right Arithmetic Variable
SRL	Shift Word Right Logical
SRLV	Shift Word Right Logical Variable

Table 3-5 CPU Logical Instructions

Mnemonic	Instruction
AND	And
ANDI	And Immediate
LUI	Load Upper Immediate
NOR	Not Or
OR	Or
ORI	Or Immediate
XOR	Exclusive Or
XORI	Exclusive Or Immediate

Table 3-6 CPU Move Instructions

Mnemonic	Instruction
MFHI	Move From HI Register
MFLO	Move From LO Register
MOVF	Move Conditional on Floating Point False
MOVN	Move Conditional on Not Zero
MOVT	Move Conditional on Floating Point True
MOVZ	Move Conditional on Zero
MTHI	Move To HI Register
MTLO	Move To LO Register

Instruction format

Table 4-23 CPU Instruction Format Fields

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM
immediate	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary *opcode* SPECIAL

Figure 4-1 Immediate (I-Type) CPU Instruction Format

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16

Figure 4-2 Jump (J-Type) CPU Instruction Format

31 26	25 0
opcode	instr_index
6	26

Figure 4-3 Register (R-Type) CPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

MIPS Core Block Diagram

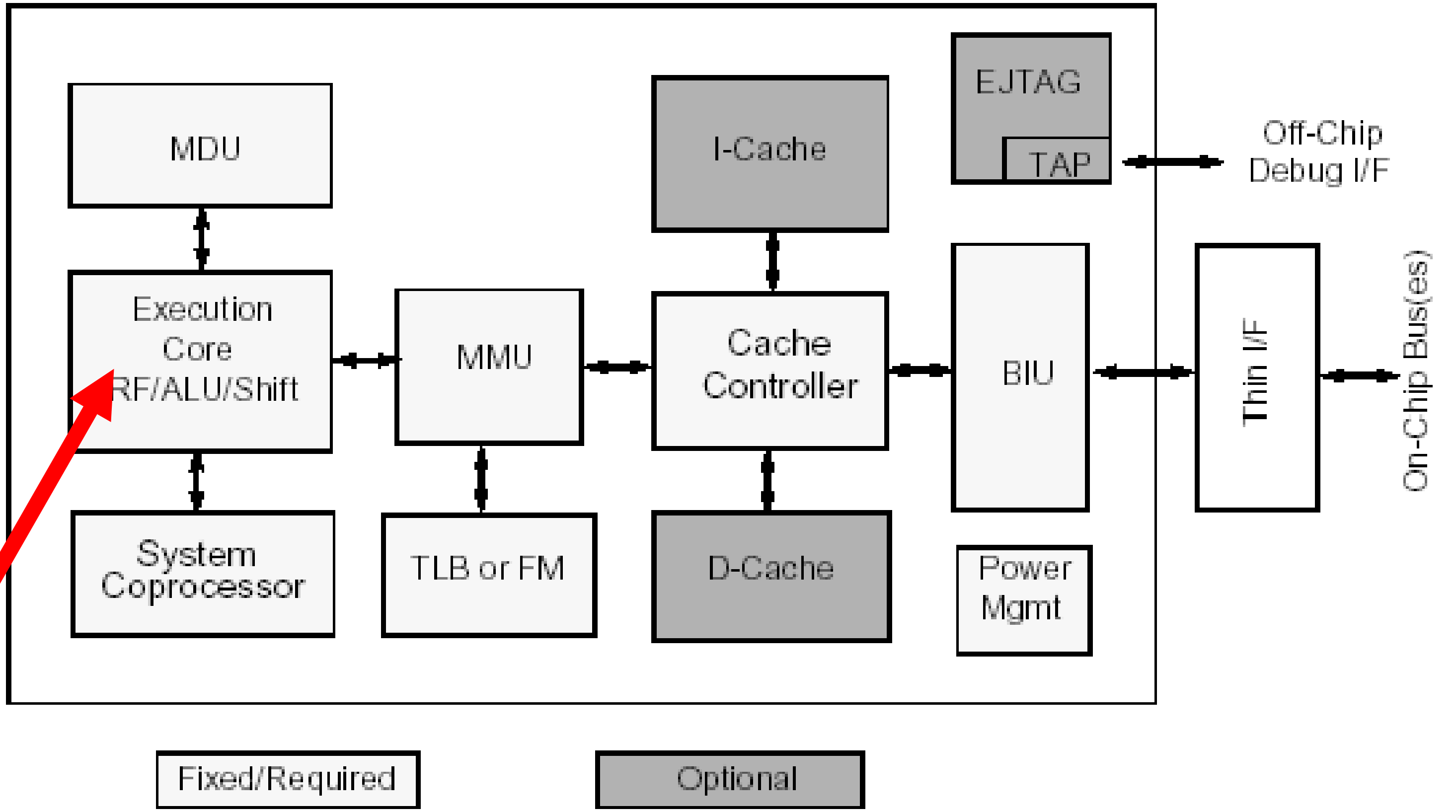
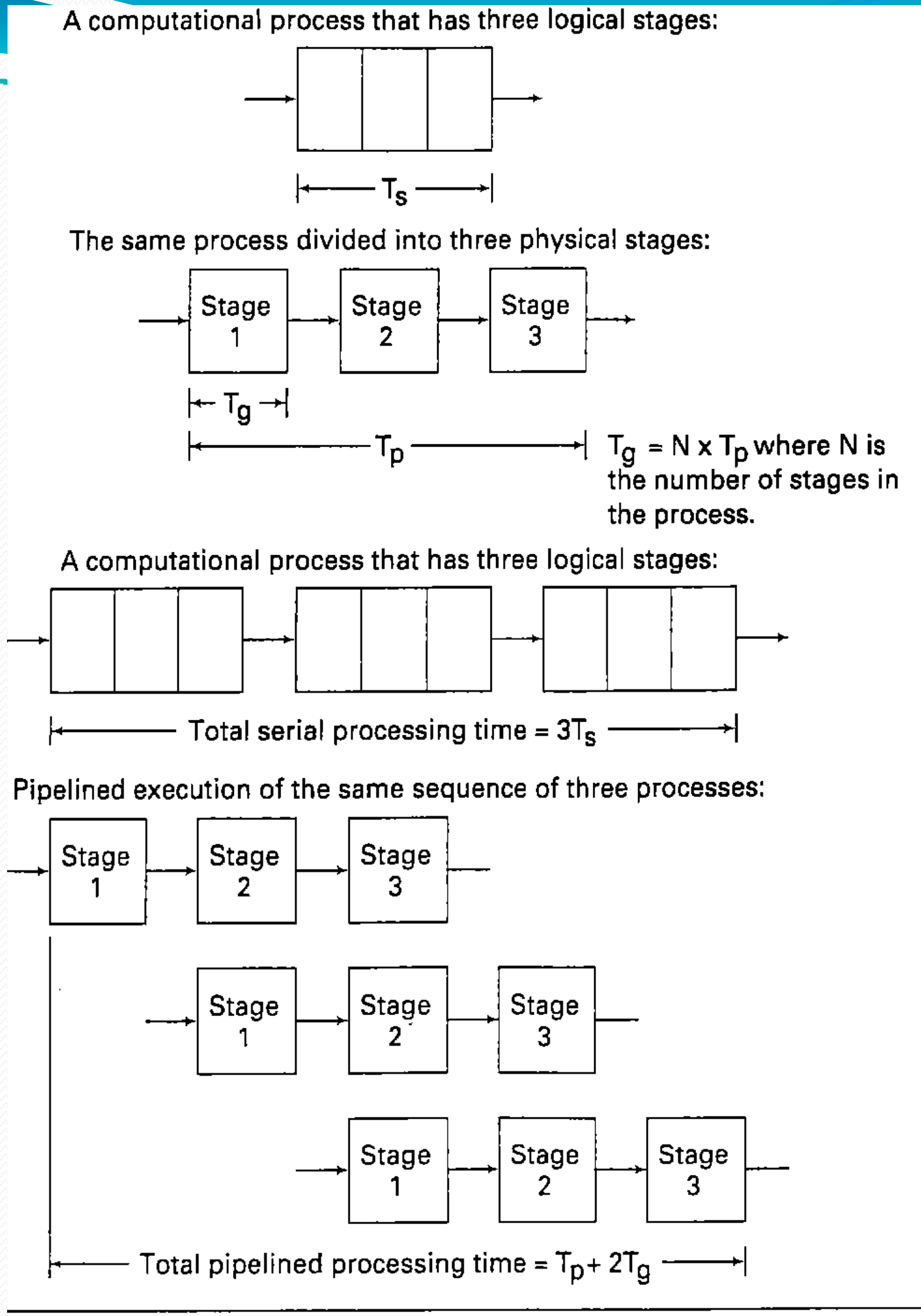


Figure 1-1 4K Processor Core Block Diagram

Looks at design of
this block in this section.

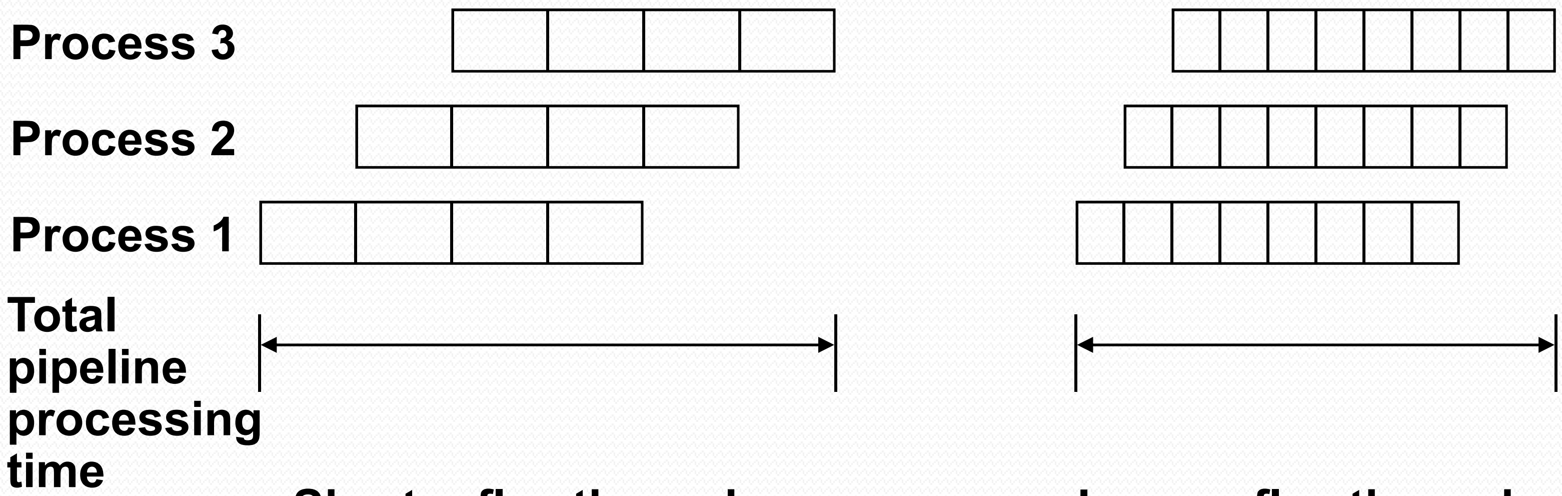
Pipelining

- Decomposition of process into stages.
- Serial and pipelined execution of the same process.
- Result is that total time taken is lesser than the sum of the individual times.



Pipeline Granularity

Coarse granularity | **Fine granularity**



Coarse granularity

- Shorter flowthrough time due to less stages (less latches)
- Longer total pipeline processing time due to longer processing time for one stage

Fine granularity

- Longer flowthrough time due to more stages (more latches)
- Shorter total pipeline processing time due to shorter processing time for one stage

Example

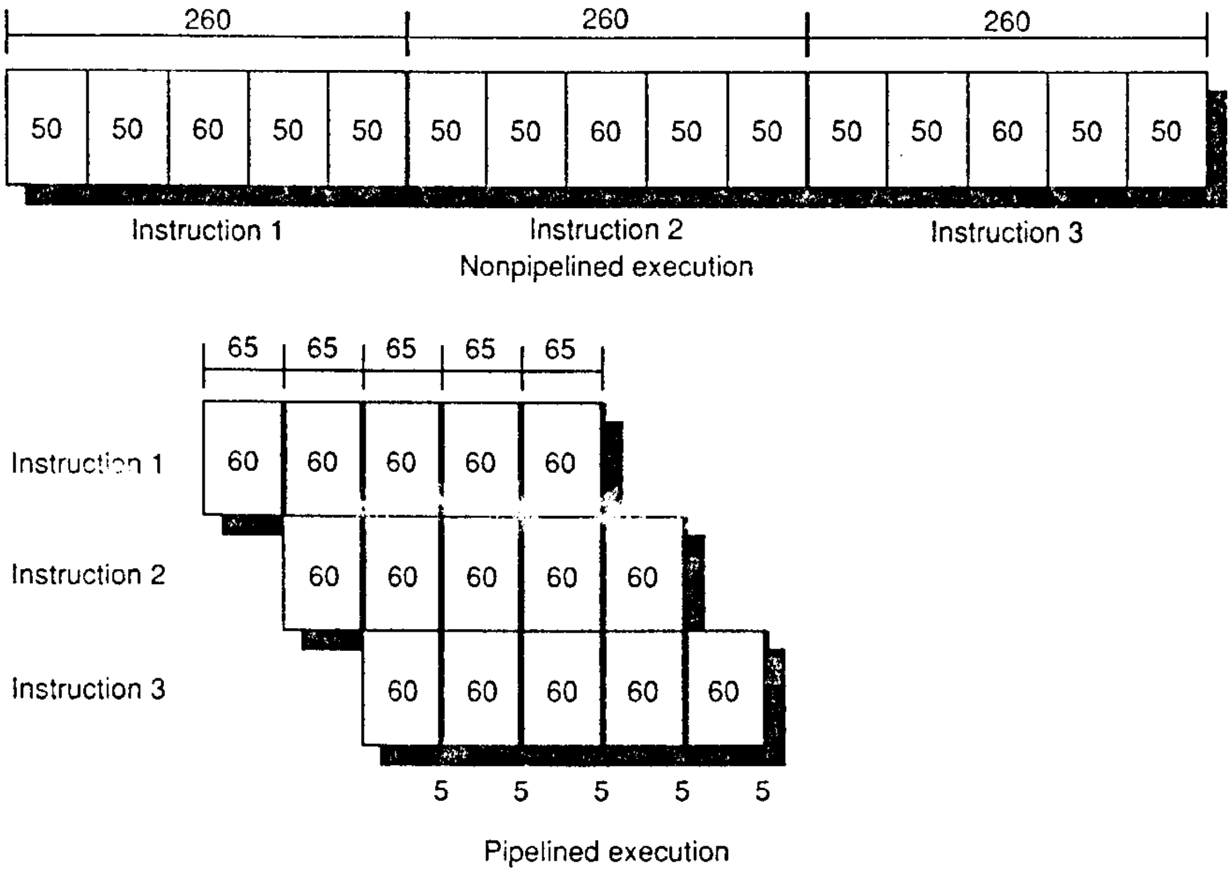
- Average instruction execution time = 50+50+60+50+50 us = 260 ns

In the nonpipelined version, the 3 instructions are executed sequentially.

In the pipelined version, the shaded areas represent the overhead of 5 ns per pipestage.

The length of the pipestages must all be the same: 60 ns plus the 5 ns overhead.

The latency of an instruction increases from 260 ns in the nonpipelined machine to 325 ns in the pipelined machine.



Parallel Pipeline

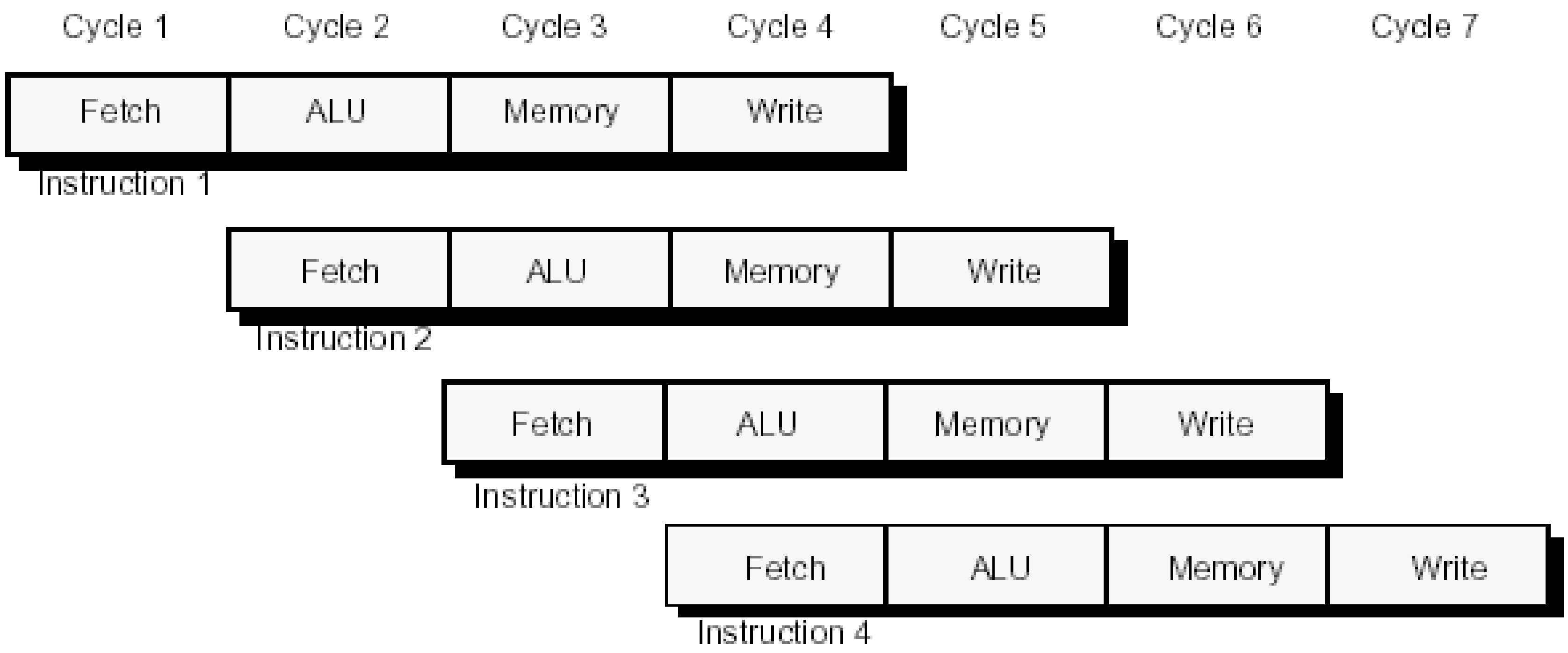


Figure 2-3 Four-Deep Single-Completion Pipeline

Super pipeline

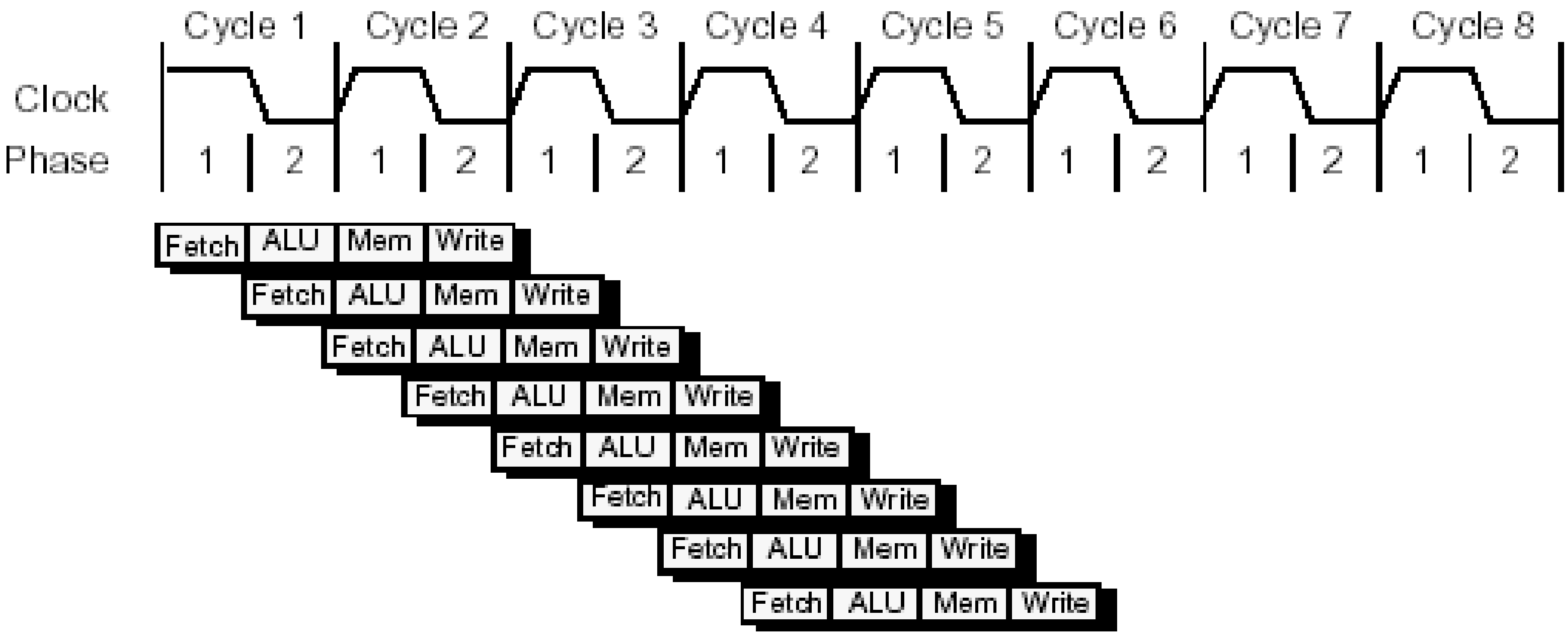
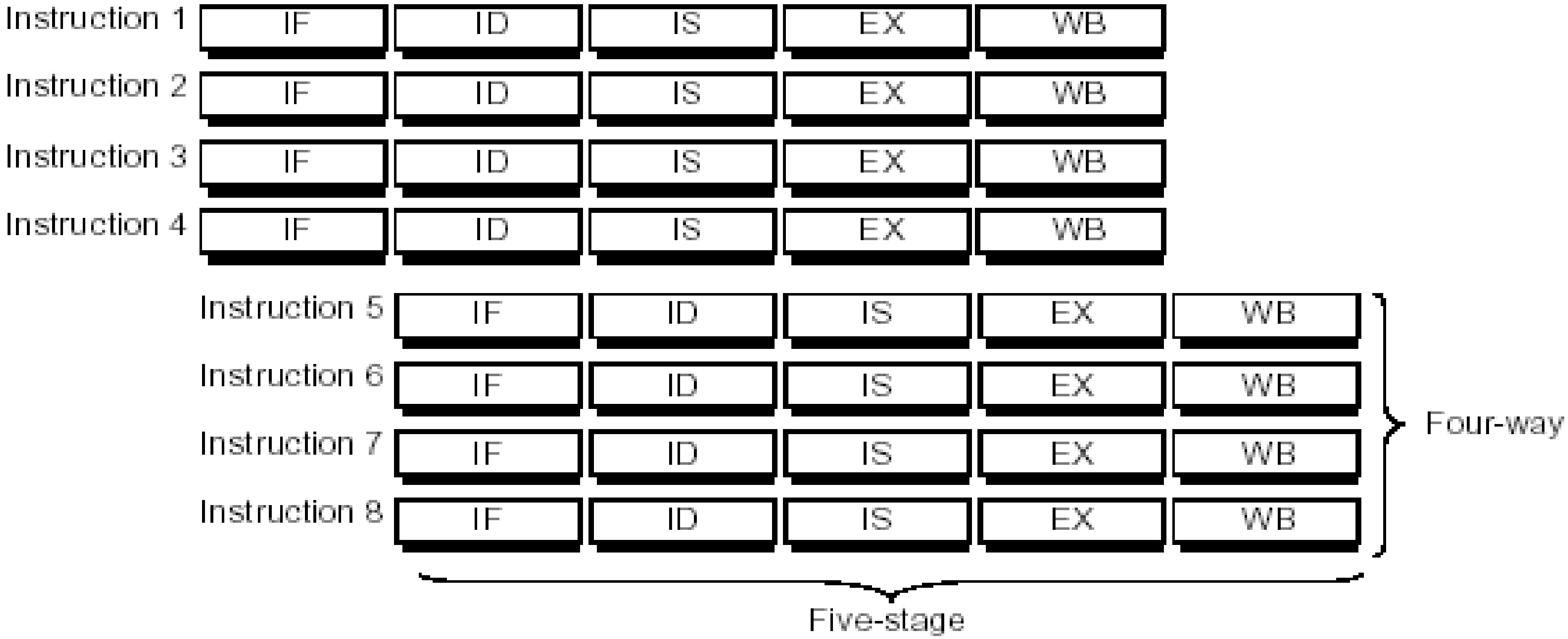


Figure 2-4 Four-Deep Superpipeline

Superscalar Pipeline



IF = instruction fetch
ID = instruction decode and dependency
IS = instruction issue
EX = execution
WB = write back

Figure 2-5 Four-Way Superscalar Pipeline

Pipeline speedup

$$\text{Pipeline speedup} = \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}}$$

$$= \frac{\text{CPI without pipelining} * \text{Clock cycle without pipelining}}{\text{CPI with pipelining} * \text{Clock cycle with pipelining}}$$

$$= \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{CPI without pipelining}}{\text{CPI with pipelining}}$$

$$\text{Ideal CPI} = \frac{\text{CPI without pipelining}}{\text{Pipeline depth}}$$

Rearranging this and substituting into the speedup equation yields:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{CPI with pipelining}}$$

If we confine ourselves to pipeline stalls,

$$\text{CPI with pipelining} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

We can substitute and obtain:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles}}$$

Pipeline hazards

- Situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
  - Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - Data hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - Control hazards arise from the pipelining of branches and other instructions that change the PC.

Structural hazard

- When a functional unit needs to be used by different stages of different instructions. Not enough duplication.

A pipeline stalled for a structural hazard—a load with one memory port.
With only one memory port, the pipeline cannot initiate a data fetch and instruction fetch in the same cycle. A load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall - no instruction is initiated on clock cycle 4 (which normally would be instruction i+3). Because the instruction being fetched is stalled, all other instructions in the pipeline can proceed normally. The stall cycle will continue to pass through the pipeline.

Instruction	Clock cycle number 1	2	3	4	5	6	7	8	9
Load instruction	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				stall	IF	ID	EX	MEM	WB
Instruction *i*+4						IF	ID	EX	MEM

Example

Suppose that data references constitute 30% of the mix and that the ideal CPI of the pipelined machine, ignoring the structural hazard, is 1.2. Disregarding any other performance losses, how much faster is the ideal machine without the memory structural hazard, versus the machine with the hazard?

The ideal machine will be faster by the ratio of the speedup of the ideal machine over the real machine. Since the clock rates are unaffected, we can use the following for speedup:

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI * Pipeline depth}}{\text{Ideal CPI + Pipeline stall cycles}}$$

Since the ideal machine has no stalls, it's speedup is simply $\frac{1.2*\text{Pipeline depth}}{1.2}$.

The speedup of the real machine is $\frac{1.2*\text{Pipeline depth}}{1.2+0.3*1} = \frac{1.2*\text{Pipeline depth}}{1.5}$.

$$\frac{\text{Speedup}_{\text{ideal}}}{\text{Speedup}_{\text{real}}} = \frac{\left(\frac{1.2*\text{Pipeline depth}}{1.2}\right)}{\left(\frac{1.2*\text{Pipeline depth}}{1.5}\right)} = \frac{1.5}{1.2} = 1.25$$

Thus, the machine without the structural hazard is 25% faster.

Data hazard

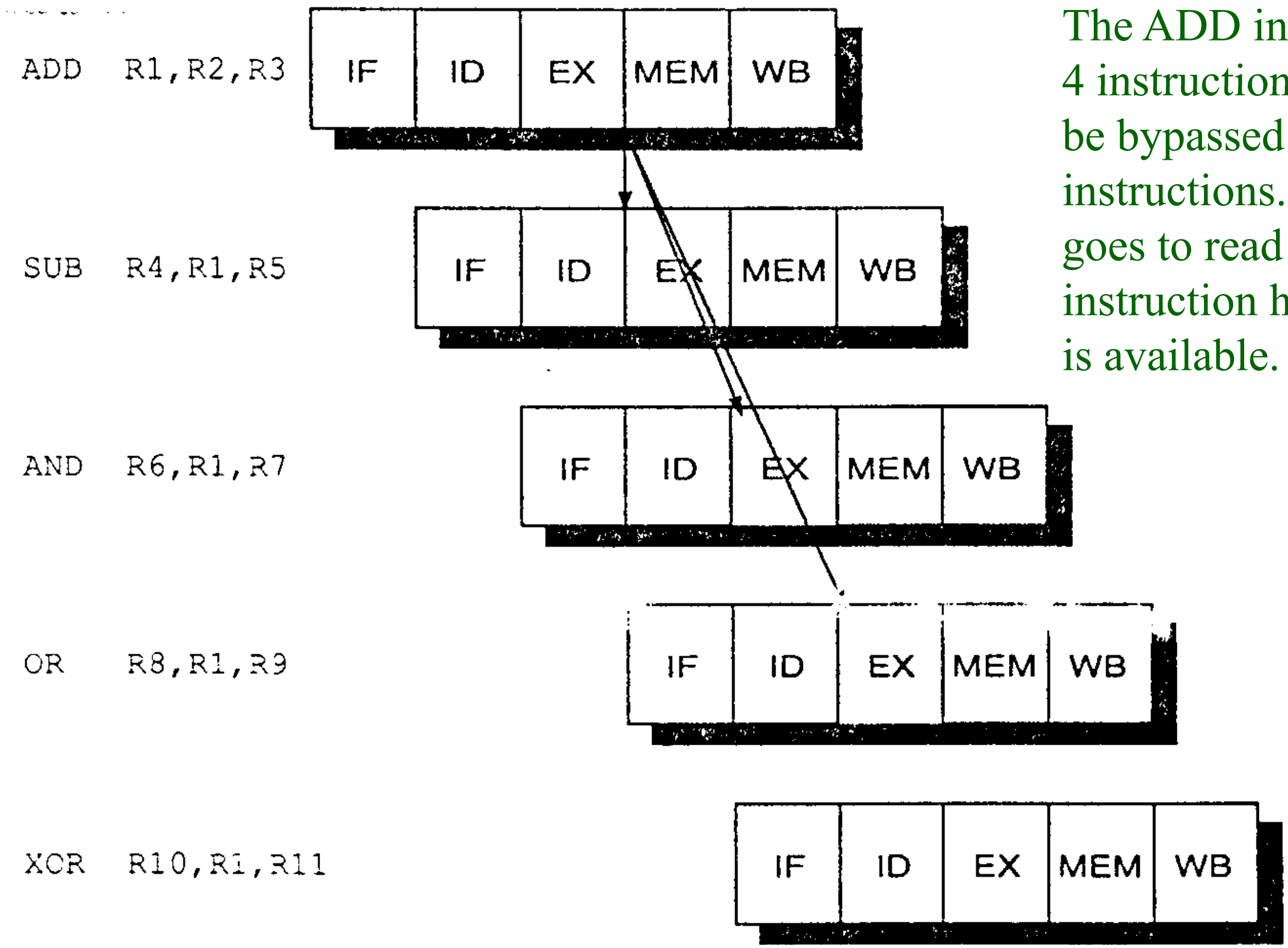
- ADD R1, R2 , R3
- SUB R4 , R1, R5

The ADD instruction writes a register that is a source operand for the SUB instruction. But the ADD doesn't finish writing the data into the register file until three clock cycles after SUB begins reading it!

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
ADD instruction	IF	ID	EX	MEM	WB–data written here	
SUB instruction		IF	ID–data read here	EX	MEM	WB

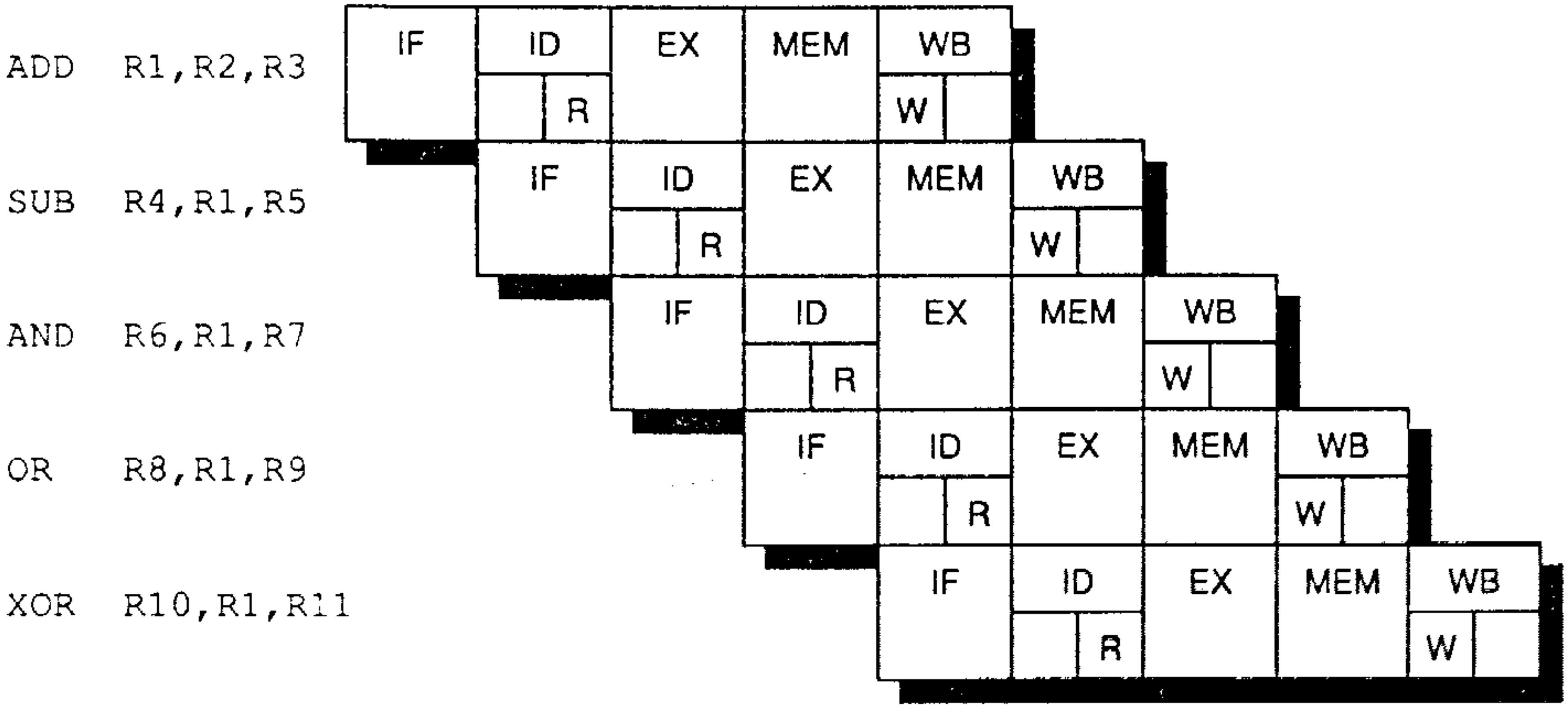
- Solution
  - hardware forwarding or bypassing
    - The ALU result is always fed back to the ALU input latches.
    - If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Instructions in pipeline that need to forward results



The ADD instruction sets R1, and the next 4 instructions use it. The value of R1 must be bypassed to the SUB, ADD, and OR instructions. By the time the XOR instruction goes to read R1 in the ID phase, the ADD instruction has completed WB, and the value is available.

The same instruction sequence as before, with register reads and writes occurring in opposite halves of the ID and WB stages



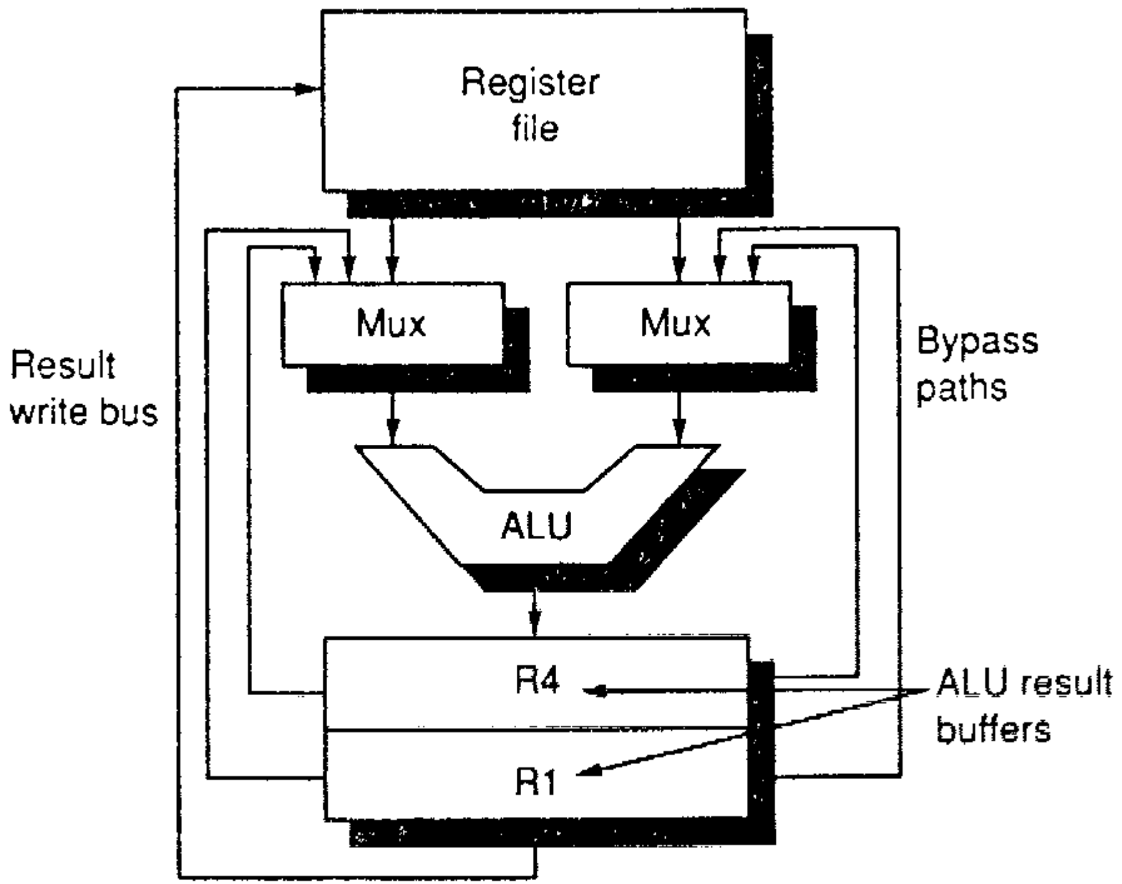
The SUB and AND instructions require the value of R1 to be bypassed to them, and this will happen as they enter their EX stage. However, by the time of the OR instruction, which also uses R1, the write of R1 has completed, and no forwarding is required. The XOR depends on the ADD, but the value of R1 from the ADD is always written back the cycle before XOR reaches its ID stage and reads it.

ALU with bypass unit

The contents of the buffer are shown at the point where the AND instruction of the code sequence in Figure (previous slide) is about to begin the EX stage.

The ADD instruction that computed R1 (in the second buffer) is in its WB stage, and the left input multiplexer is set to pass the just computed value of R1 (not the value read from the register file) as the first operand to the AND instruction.

The result of the subtract, R4, is in the first buffer.



Data hazards

- Dependence between instructions such that when overlapped in execution, changed order of access.
  - Register
  - memory
    - cache misses can cause problems if processor allowed to proceed.
- Types
  - **RAW** *(read after write)—j* tries to read a source before i writes it, so j incorrectly gets the old value.
  - **WAR** *(write after read)—j* tries to write a destination before it is read by i, so i incorrectly gets the new value. This cannot happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source after a write of an instruction later in the pipeline. For example, auto-increment addressing can create a WAR hazard.
  - ***WAW*** *(write after write)—j* tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by *j* in the destination. This hazard is present only in pipelines that write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled). The DI,X pipeline writes a register only in WE and avoids this class of hazards.

Unavoidable delays with loads

Pipeline hazard occurring when the result of a load instruction is used by the next instruction as a source operand and is forwarded. The value is available when it returns from memory at the end of the load instruction's MEM cycle. However, it is needed at the beginning of that clock cycle for the ADD (the EX stage of the add).

The load value can be forwarded to the SUB instruction and will arrive in time for that instruction (EX). The AND can simply read the value during ID since it reads the registers in the second half of the cycle and the value is written in the first half.

LW R1,32(R6)	IF	ID	EX	MEM	WB
ADD R4,R1,R7		IF	ID	EX	MEM
SUB R5,R1,R8			IF	ID	**EX**
AND R6,R1,R7				IF	**ID**

The effect of the stall on the pipeline.

All instructions starting with the instruction that has the dependence are delayed.

With the delay, the value of the load that returns in MEM can now be forwarded to the EX cycle of the ADD instruction.

Because of the stall, the SUB instruction will now read the value from the registers during its ID cycle rather than having it forwarded from the MDR.

Any instruction	IF	ID	EX	MEM	WB				
LW R1,32(R6)		IF	ID	EX	**MEM**	WB			
ADD R4,R1,R7			IF	ID	*stall*	**EX**	MEM	WB	
SUB R5,R1,R8				IF	*stall*	**ID**	EX	MEM	WB
AND R6,R1,R7					*stall*	IF	ID	EX	MEM	WB

Pipeline interlock

Example of A=B+C

The ADD instruction must be stalled to allow the load of C to complete.

LW R1,B	IF	ID	EX	MEM	WB				
LW R2,C		IF	ID	EX	MEM	WB			
ADD R3,R1,R2			IF	ID	stall	EX	MEM	WB	
SW A,R3				IF	stall	ID	EX	MEM	WB

The SW need not be delayed further because the forwarding hardware passes the result from the ALU directly to the MDR for storing.

Performance of scheduling

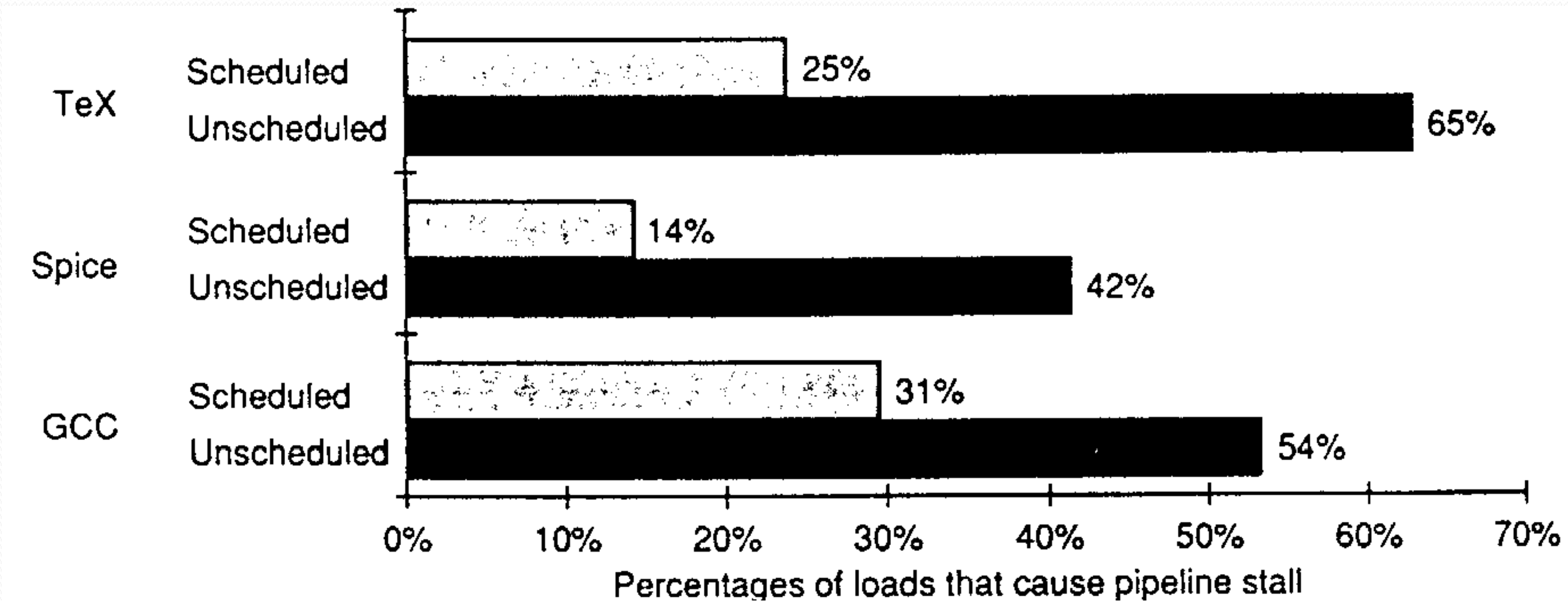
- Instruction scheduling to fill up slack
- depends on software to avoid the hazard
- delay load: a load requiring that the following do not use its result.

Percentage of the loads that result in a stall with the DLX pipeline.

Black bars show the amount without compiler scheduling.

Gray bars show the effect of a good, but simple, scheduling algorithm.

These data show scheduling effectiveness after global optimization. Global optimization actually makes scheduling relatively harder because there are fewer candidates available for scheduling into delay slots. For example, on GCC and TeX, when the programs are scheduled but not globally optimized, the percentage of load delays that result in a stall drops to 22% and 19%, respectively)



Example

Suppose that 20% of the instructions are loads, and half the time the instruction following a load instruction depends on the result of the load. If this hazard creates a single cycle delay, how much faster is the ideal pipelined machine (with a CPI of 1) that does not delay the pipeline, compared to a more realistic pipeline? Ignore any stalls other than pipeline stalls.

The ideal machine will be faster by the ratio of the CPIs. The CPI for an instruction following a load is 1.5, since they stall half the time. Since loads are 20% of the mix, the effective CPI is $(0.8*1 + 0.2*1.5) = 1.1$. This yields a performance ratio of 1.1 / 1. Hence, the ideal machine is 10% faster.

Example

Q: Generate DLX code that avoids pipeline stalls for the following sequence:
a = b + c;
d = e – f;
Assume loads have a latency of one clock cycle.

Scheduled code:

```
LW      Rb,b
LW      Rc , c
LW       Re, e
; swapped with next instruction to avoid stall
ADD     Ra,Rb,Rc
LW      Rf, f
SW      a, Ra
; store/load interchanged to avoid stall in SUB
SUB     Rd. Re, Rf
SW      d,Rd
```

Both load interlocks (LW Rc,c/ADD Ra,Rb,Rc and LW Rf,f/SUB Rd. Re, Rf) have been eliminated. There is a dependence between the ALU instruction and the store, but the pipeline structure allows the result to be forwarded. Notice that the use of different registers for the first and second statements was critical for this schedule to be legal. In particular, if the variable e were loaded into the same register as b or c, this schedule would not be legal. In general, pipeline scheduling can increase the register count required.

Implementing Data Hazard Detection in Simple Pipelines

- Hardware requirement
  - Additional multiplexers on the inputs to the ALU Just as was required for the bypass hardware for register-register instructions)
  - Extra paths from the MDR to both multiplexer inputs to the ALU
  - A buffer to save the destination-register numbers from the prior two instructions (the same as for register-register forwarding)
  - Four comparators to compare the two possible source register fields with the destination fields of the prior instructions and look for a match

Data dependencies

Situation	Example code sequence	Action

No dependence	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	`LW R1,45(R2)`
	`ADD R5,R1,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.

Dependence overcome by forwarding	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.

Dependence with accesses in order	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R1,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half. See Figure 6.8 (page 262).

FIGURE 6.14 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only compare needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case, once execution continues.

Control hazards

Branch instruction	IF	ID	EX	MEM	WB					
Instruction *i*+1		*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB	
Instruction *i*+2			*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB
Instruction *i*+3				*stall*	*stall*	*stall*	IF	ID	EX	MEM
Instruction *i*+4					*stall*	*stall*	*stall*	IF	ID	EX
Instruction *i*+5						*stall*	*stall*	*stall*	IF	ID
Instruction *i*+6							*stall*	*stall*	*stall*	IF

FIGURE 6.15 Ideal DLX pipeline stalling after a control hazard. The instruction labeled instruction *i+k* represents the *k*th instruction executed after the branch. There is a difficulty in that the branch instruction is not decoded until after instruction *i* +1 has been fetched. This figure shows the conceptual difficulty, while Figure 6.16 shows what really happens.

Branch instruction	IF	ID	EX	MEM	WB					
Instruction *i*+1		IF	*stall*	*stall*	IF	ID	EX	MEM	WB	
Instruction *i*+2			*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB
Instruction *i*+3				*stall*	*stall*	*stall*	IF	ID	EX	MEM
Instruction *i*+4					*stall*	*stall*	*stall*	IF	ID	EX
Instruction *i*+5						*stall*	*stall*	*stall*	IF	ID
Instruction *i*+6							*stall*	*stall*	*stall*	IF

FIGURE 6.16 What might really happen in the DLX pipeline. Instruction *i* +1 is fetched, but the instruction is ignored and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for instruction *i* +1 is redundant. This will be addressed shortly.

Reduction of pipeline penalty

- Find out whether the branch is taken or not earlier in the pipeline.
- Compute the taken PC (address of the branch target) earlier.
- Both to be used
- For simple conditions, possible by end of ID phase

Revised pipeline structure, use of a separate adder to compute the branch target address.

Branch target address (BTA) addition happens during ID for all instructions.

Branch condition (Rs1 op 0) will also be done for all instructions. The last operation in ID is to replace the PC. We must know that the instruction is a branch before we perform this step. This requires decoding the instruction before the end of ID, or doing this operation at the very beginning of EX when the PC is sent out.

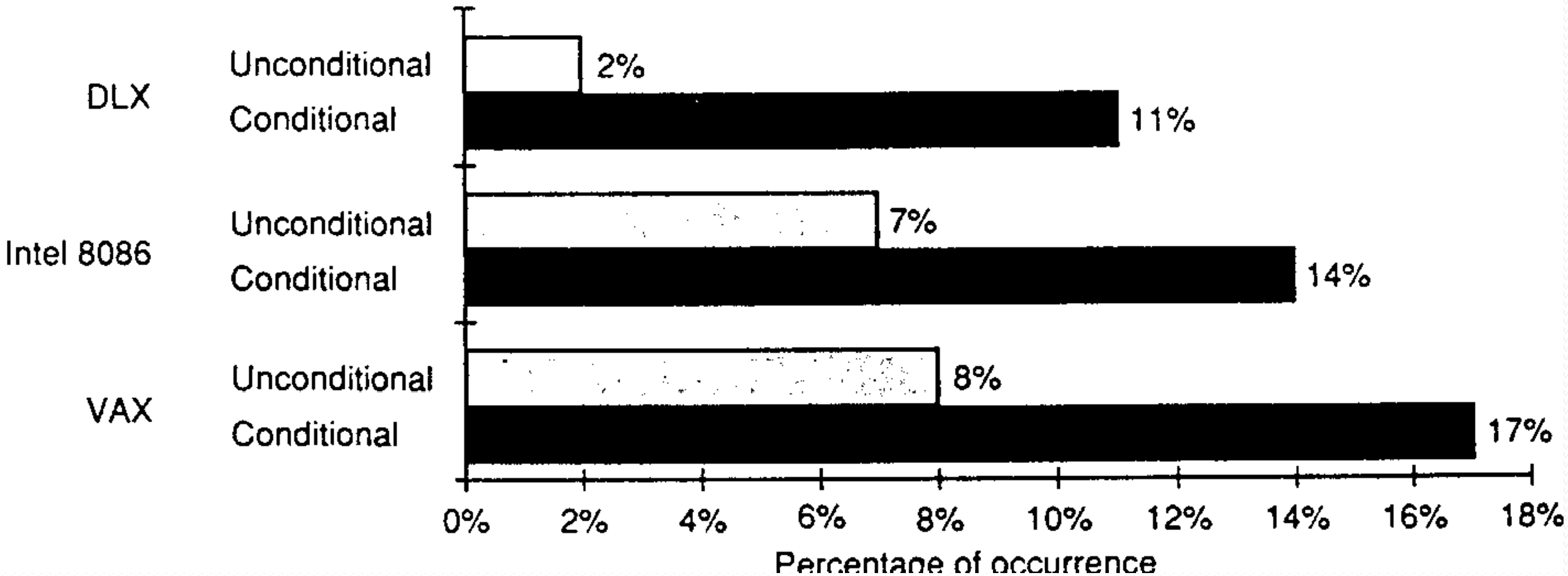
Because the branch is done by the end of ID, the EX, MEM, and WB stages are unused for branches.

An additional complication arises for jumps that have a longer offset than branches. We can resolve this by using an additional adder that sums the PC and lower 26 bits of the IR. Alternatively, we could attempt a clever scheme that does a 16 bit add in the first half of the cycle and determines whether to add in 10 bits from IR in the second half of the cycle, by decoding the jump opcodes early.

Pipe stage	**Branch instruction**
IF	`IR←Mem[PC];` `PC←PC+4;`
ID	`A← Rs1;    B← Rs2;` **$BTA \leftarrow PC+((IR_{16})^{16}\#\#\ IR_{16..31})$** **`if (Rs1 op 0) PC←BTA`**
EX	
MEM	
WB	

Branch Behavior in Programs

The frequency of instructions (branches, jumps, calls, and returns) that may change the PC



These data represent the average over various programs.

Instructions are divided into two classes: branches, which are conditional (including loop branches), and those that are unconditional bumps, calls, and returns).

Reducing Pipeline Branch Penalties

predict-not-taken scheme

Untaken branch instruction	IF	**ID**	EX	MEM	WB				
Instruction *i*+1		**IF**	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB

Pipeline sequence when the branch is untaken

Taken branch instruction	IF	**ID**	EX	MEM	WB				
Instruction *i*+1		**IF**	**IF**	ID	EX	MEM	WB		
Instruction *i*+2			*stall*	IF	ID	EX	MEM	WB	
Instruction *i*+3				*stall*	IF	ID	EX	MEM	WB
Instruction *i*+4					*stall*	IF	ID	EX	MEM

Pipeline sequence when the branch is taken

When the branch is untaken, determined during ID, we have fetched the fall through and just continue.

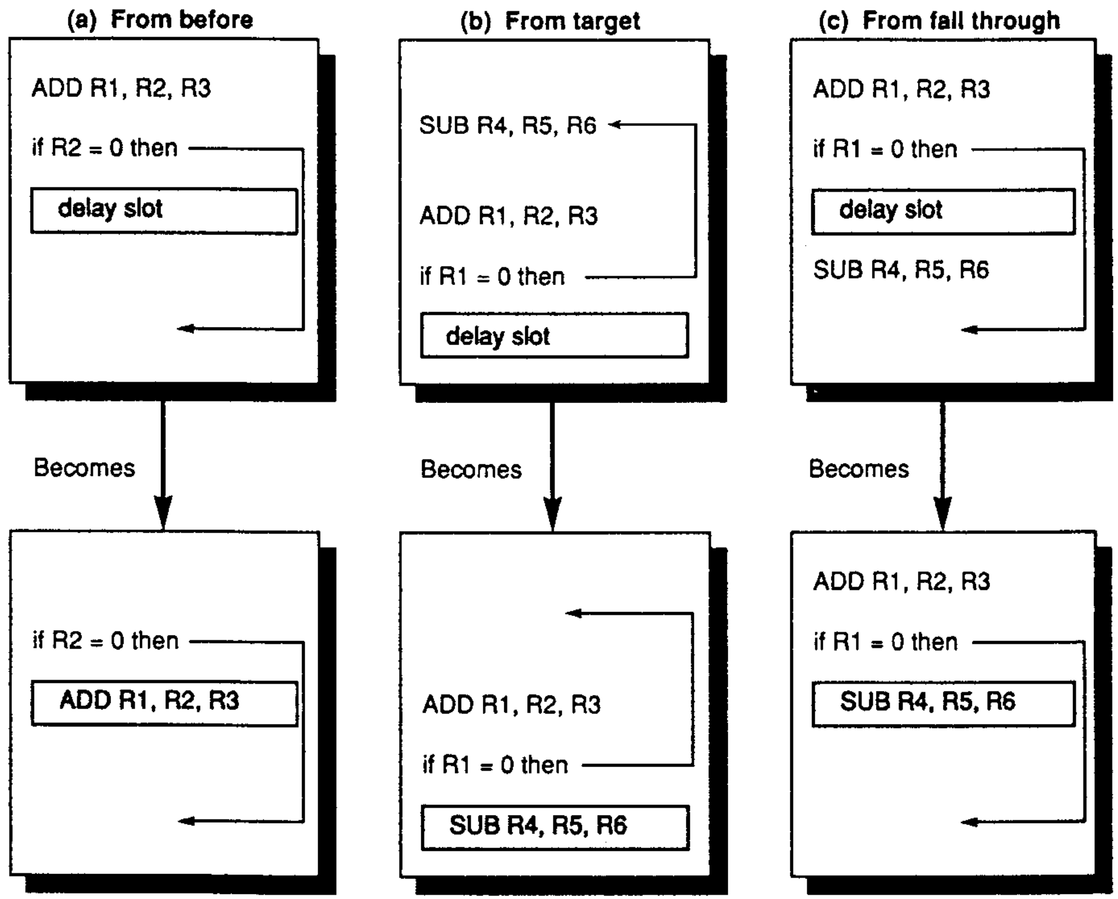
If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

Reducing Pipeline Branch Penalties

Scheduling the branch delay slot

branch instruction
sequential successor1
sequential successor2
……..
sequential successor3
branch target if taken

Before scheduling



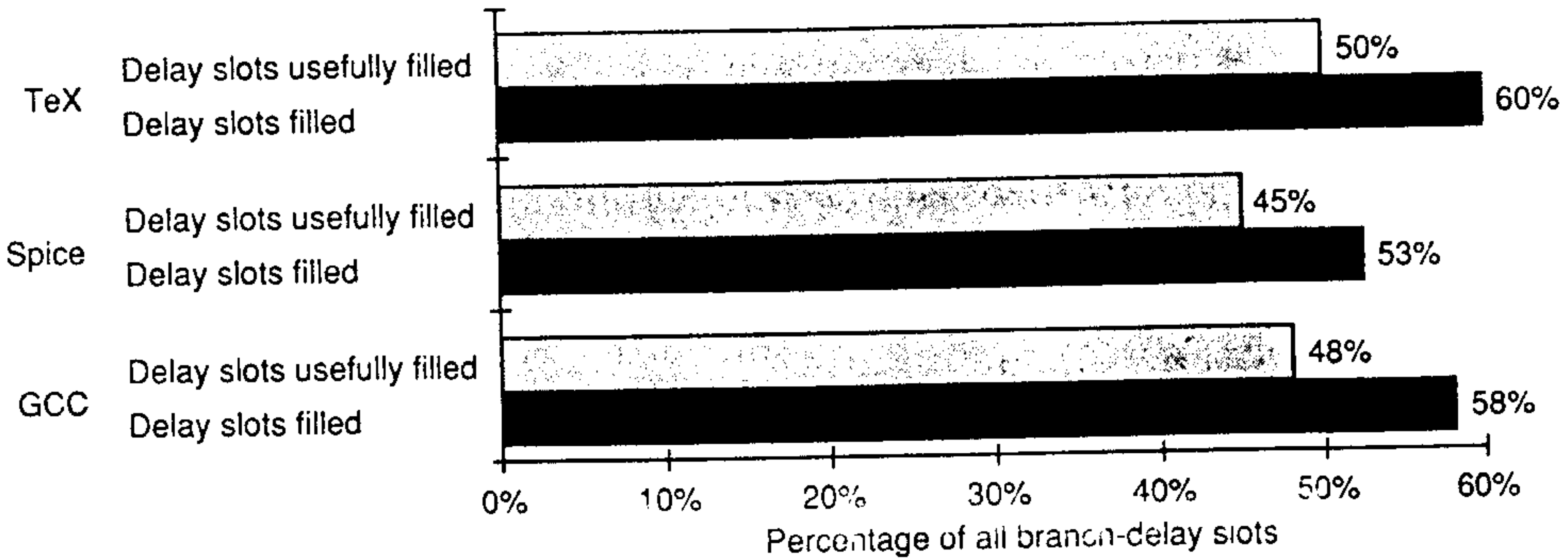
After scheduling

Delayed branch / slot

Different constraints for each of these branch-scheduling schemes, as well as situations in which they win.

Scheduling strategy	Requirements	Improves performance when?
(a) From before branch	Branch must not depend on the rescheduled instructions.	Always.
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
(c) From fall through	Must be OK to execute instructions if branch is taken.	When branch is not taken.

Performance of delayed branch



Effective performance of schemes

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycle}}$$

If we assume that the ideal CPI is 1, then we can simplify this:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Since Pipeline stall cycles from branches = Branch frequency * Branch penalty

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{(1 + \text{Branch frequency} * \text{Branch penalty})}$$

Some measurements

Scheduling scheme	Branch penalty	Effective CPI	Pipeline speedup over nonpipelined machine	Pipeline speedup over stall pipeline on branch
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Dealing with Interrupts

- Difficult to know when an instruction can safely change state of pipeline
  - Interrupt can force abort of instruction before completion
- More difficult if:
  - occur within instruction, eg page fault
  - must be restartable
- Needs to save pipeline on interrupt and restart pipeline later
  - Action to save pipeline on interrupt:
    - Force a trap instruction into the pipeline on the next IF.
    - Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline. This prevents any state changes for instructions that will not be completed before the interrupt is handled.
    - After the interrupt handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the interrupt later.
  - Re-start by restoring PC

Dealing with Interrupts

- Delay Branch (instructions not sequentially related)
  - need to save a number of PC; one more than the length of branch delay
- Precise Interrupts
  - pipeline stopped
  - instructions before faulting instruction are complete
  - instructions after faulting instruction can be restarted
- Precise interrupts - a requirement for many systems
  - demand paging
  - IEEE arithmetic trap handlers
- Difficult to implement
  - may need to undo some state changes

Where can interrupts occur ?

Pipeline stage	Problem interrupts occurring
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation
WB	None

Dealing with Interrupts

Interrupts in the same clock cycle.
Deal with page fault, then exit.
Arithmetic will interrupt again, enter again and deal.

Page fault here

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Arithmetic interrupt here

Later instruction interrupts first

Page fault here

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Instruction page fault here

First approach

- Precise approach
  - hardware post interrupt in a status vector for each instruction
  - check status when enters WB state
  - process pending interrupts in order they would occur in time
- Actions on behalf of instruction i or later : maybe invalid
  - not a problem as state are only changed in WB
  - improvement
    - disable action on behalf of instructions on interrupt detect

Second approach

Instruction *i*–3	IF	ID	EX	MEM	WB					
Instruction *i*–2		IF	ID	EX	MEM	WB				
Instruction *i*–1			IF	ID	EX	MEM	WB			
Instruction *i* (LW)				IF	ID	EX	MEM	WB		
Instruction *i*+1 (ADD)					**IF**	ID	EX	MEM	WB	
Instruction *i*+2						IF	ID	EX	MEM	WB

Instruction *i*–3	IF	ID	EX	MEM	WB					
Instruction *i*–2		IF	ID	EX	MEM	WB				
Instruction *i*–1			IF	ID	EX	MEM	WB			
Instruction *i* (LW)				IF	ID	EX	**MEM**	WB		
Instruction *i*+1 (ADD)					IF	ID	EX	MEM	WB	
Instruction *i*+2						IF	ID	EX	MEM	WB
Instruction *i*+3							IF	ID	EX	MEM
Instruction *i*+4								IF	ID	EX

Handle interrupt immediately.
Stop pipeline without complete instructions that have yet to change state.

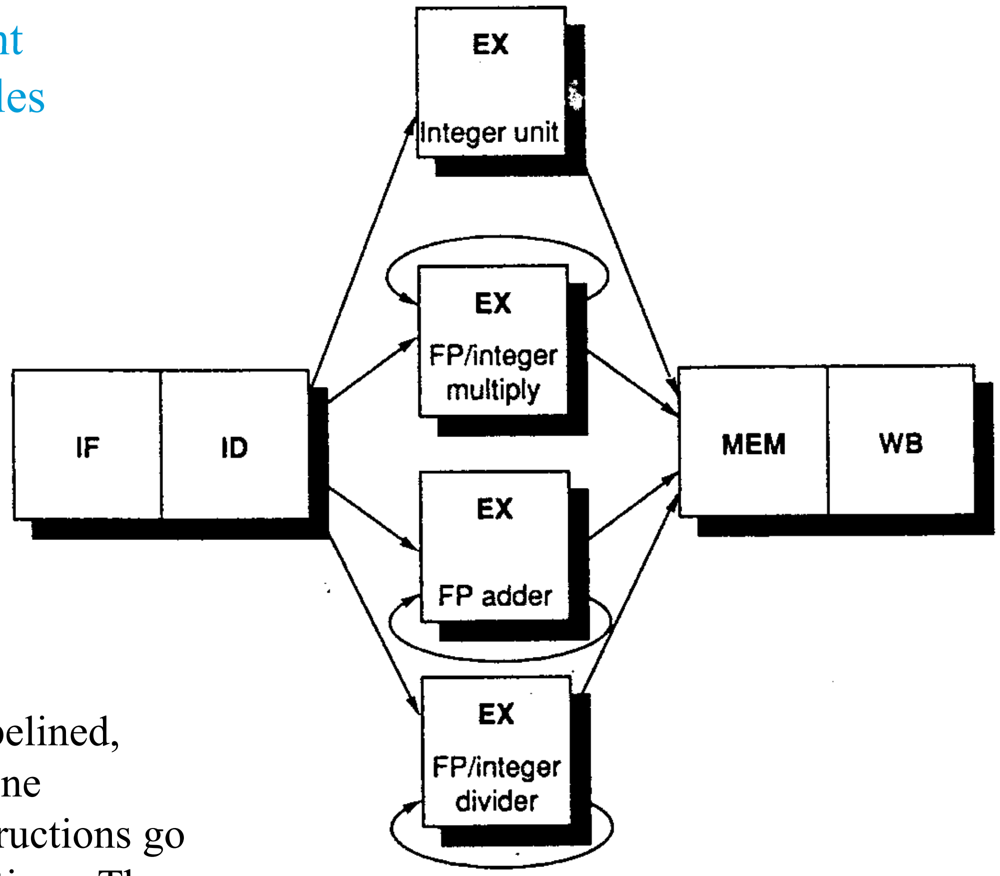
Top: i-2, i-1, i, i+1

Bottom: i, … i+3

Multicycle operations

Different instructions in particular floating point instructions take different number of clock cycles to complete.

EX cycle may be repeated many times before completion of instruction



The DLX pipeline with three additional nonpipelined, floating point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating point operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

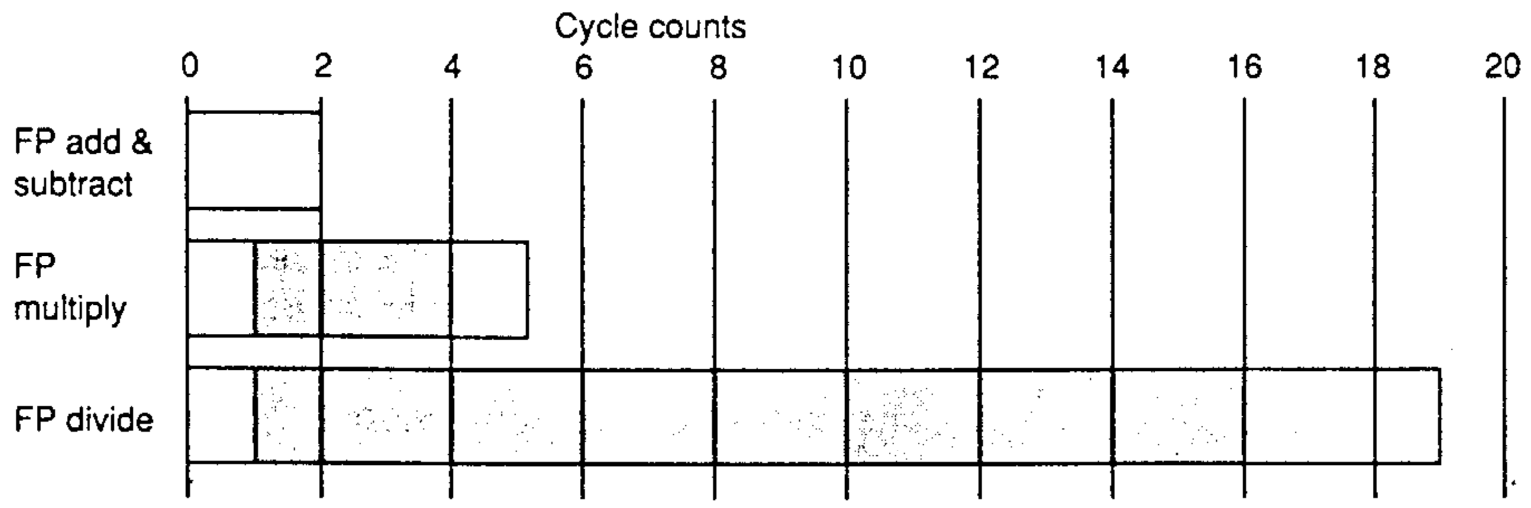
Types of hazards

- RAW
  - resolve by
    - Check for structural hazard—Wait until the required functional unit is not busy.
    - Check for a RAW data hazard—Wait until the source registers are not listed as destinations by ally of the EX stages in the functional units.
    - Check for forwarding—Test if the destination register of an instruction in MEM or WB is one of the source registers of the floating point instruction; if so, enable the input multiplexer to use that result, rather than the register contents.
- Contention for register access (to write) at the end of the pipeline
  - resolved by
    - establishing a static priority for use of the WB stage. If multiple instructions wish to enter the MEM stage simultaneously, all instructions except the one with the highest priority are stalled in their EX stage.

Types of hazards

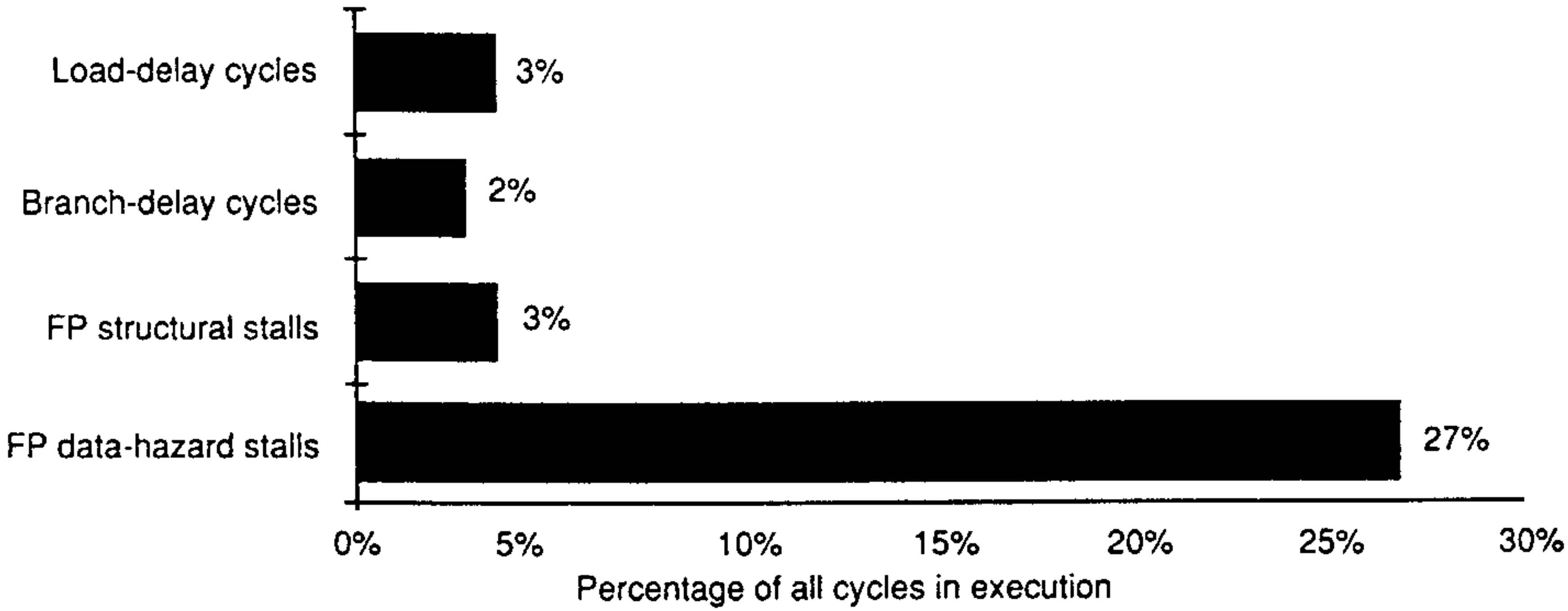
- Ensuring precise interrupts
  - out of order completion due to different execution times
    - DIVF F0,F2,F4
    - ADDF Fl0,Fl0,F8
    - SUBF F12,F12,F14
      - eg if DIVF has an interrupt, ADDF may have completed and modified F10, so that it is not possible to restore the state
  - action
    - to ignore the problem and settle for imprecise interrupts
    - queue the results of an operation until all the operations that were issued earlier are complete
    - to allow the interrupts to become somewhat imprecise, but keep enough information so that the trap handling routines can create a precise sequence for the interrupt
    - allows the instruction issue to continue only if it is certain that all the instructions before the issuing instruction will complete without causing an interrupt

Performance of FP pipeline



Total clock cycle count and permissible overlap among double precision, floating-point operations on the MIPS R2010/3010 FP unit. The overall length of the bar shows the total number of EX cycles required to complete the operation. For example, after five clock cycles a multiply result is available. The shaded regions are times during which FP operations can be overlapped. As is common in most FP units, some of the FP logic is shared—the rounding logic, for example, is often shared. This means that FP operations with different running times cannot overlap arbitrarily. Also note that multiply and divide are not pipelined in this FP unit, so only one multiply or divide can be outstanding.

Performance of FP pipeline



Percentage of clock cycles in Spice that are pipeline stalls. This again assumes a perfect memory system with no memory system stalls. In total, 35% of the clock cycles in Spice are stalls, and without any stalls Spice would run about 50% faster.

Scheduling within the loop

Not accounting for pipeline

```
Loop: LD     F0,0(R1)     ; load the vector element
      ADDD   F4,F0,F2     ; add the scalar in F2
      SD     0(R1),F4     ; store the vector element
      SUB    R1,R1,#8     ; decrement the pointer by
                          ; 8 bytes (per DW)
      BNEZ   R1,LOOP      ; branch when it's zero
```

Execution without scheduling

			Clock cycle issued
Loop:	LD	F0,0(R1)	1
	stall		2
	ADDD	F4,F0,F2	3
	stall		4
	stall		5
	SD	0(R1),F4	6
	SUB	R1,R1,#8	7
	BNEZ	R1,LOOP	8
	stall		9

Schedule the loop

```
Loop: LD     F0,0(R1)
      stall
      ADDD   F4,F0,F2
      SUB    R1,R1,#8
      BNEZ   R1,LOOP     ; delayed branch
      SD     8(R1),F4    ; changed because interchanged with SUB
```

Execution time reduce from 9 to 6

Increasing parallelism with loop unrolling

loop unrolled 3 times (yielding four copies of the loop body),

Drop the unnecessary SUB and BNEZ operations duplicated during unrolling.

eliminated 3 branches and 3 decrements of R1. The addresses on the loads and stores have been compensated for. Without scheduling, every operation is followed by a dependent operation, and thus will cause a stall. This loop will run in 27 clock cycles—each LD takes 2 clock cycles, each ADDD 3, the branch 2, and all other instructions 1—or 6.8 clock cycles for each of the four elements.

```
Loop:  LD     F0,0(R1)
       ADDD   F4,F0,F2
       SD     0(R1),F4 ;drop SUB & BNEZ
       LD     F6,-8(R1)
       ADDD   F8,F6,F2
       SD     -8(R1),F8 ;drop SUB & BNEZ
       LD     F10,-16(R1)
       ADDD   F12,F10,F2
       SD     -16(R1),F12 ;drop SUB & BNEZ
       LD     F14,-24(R1)
       ADDD   F16,F14,F2
       SD     -24(R1),F16
       SUB    R1,R1,#32
       BNEZ   R1,LOOP
```

Unrolled and scheduled

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared to 6.8 per element before scheduling

```
Loop:  LD      F0,0(R1)
       LD      F6,-8(R1)
       LD      F10,-16(R1)
       LD      F14,-24(R1)
       ADDD    F4,F0,F2
       ADDD    F8,F6,F2
       ADDD    F12,F10,F2
       ADDD    F16,F14,F2
       SD      0(R1),F4
       SD      -8(R1),F8
       SD      -16(R1),F12
       SUB     R1,R1,#32    ;branch dependence
       BNEZ    R1,LOOP
       SD      8(R1),F16 ; 8-32 = -24
```

Superscalar computer

- Multiple instructions issued per clock cycle
  - CPI less than 1
  - one FP and one integer operation issued together
    - no need for additional hardware
    - only for FP load, store or move

Superscalar pipeline in operation. The integer and floating point instructions are issued at the same time, and each executes at its own pace through the pipeline. This scheme will only improve the performance of programs with a fair amount of floating point.

Instruction type	Pipe	Stages					
Integer instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Integer instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Integer instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	MEM	WB

Loop unrolling on a superscalar DLX

unroll with five copies of the body

Runs in 12 clock cycles per iteration, or 2.4 clock cycles per element, versus 3.5 for the scheduled and unrolled loop on the ordinary DLX pipeline.

The performance of the superscalar DLX is limited by the balance between integer and FP computation. Every FP instruction is issued together with an integer instruction, but there are not enough FP instructions to keep the FP pipeline full.

	Integer instruction		FP instruction	Clock cycle
Loop:	LD	F0,0(R1)		1
	LD	F6,-8(R1)		2
	LD	F10,-16(R1)	ADDD F4,F0,F2	3
	LD	F14,-24(R1)	ADDD F8,F6,F2	4
	LD	F18,-32(R1)	ADDD F12,F10,F2	5
	SD	0(R1),F4	ADDD F16,F14,F2	6
	SD	-8(R1),F8	ADDD F20,F18,F2	7
	SD	-16(R1),F12		8
	SD	-24(R1),F16		9
	SUB	R1,R1,#40		10
	BNEZ	R1,LOOP		11
	SD	8(R1),F20		12

When scheduled, the original loop ran in 6 clock cycles per iteration. We have improved on that by a factor of 2.5, more than half of which came from loop unrolling, which took us from 6 to 3.5, with the rest coming from issuing more than one instruction per clock cycle.

Very long instruction word (VLIW)

- Packages multiple operations in one instruction
  - have set of fields fore each functional unit
  - must have enough work in straight line code
    - unrolling and scheduling across basic blocks

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation / branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2	ADDD F24,F22,F2	
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2		
SD -16(R1),F12	SD -24(R1),F16			
SD -32(R1),F20	SD -40(R1),F24			SUB R1,R1,#48
SD -0(R1),F28				BNEZ R1,LOOP

The loop has been unrolled 6 times, which eliminates stalls, and runs in 9 cycles. This yields a running rate of 7 results in 9 cycles, or 1.28 cycles per result.

Limitation of VLIW

- Limited parallelism
  - needs to unroll loops many time to get enough independent parallel operations
  - requires “average pipeline depth x the number of functional units” independent operations
- Limited hardware
  - cost scales more than linearly
    - large increase in memory requirement
    - needs more memory port
    - increase in complexity
- Code size limitation
  - needs to unroll many loops to get independent operations
  - instruction not filled translates into wasted bits

Be Warned

If you believe everything that appears in the next 8 slides, you could be in trouble.

Reduced instruction set computer (RISC)

- **Percentage of each statement type in a measured sample of programs**

Statement	Fortran	C	Pascal	Average
Assignment	51	38	45	44.7
If	10	43	29	27.3
Call	5	12	15	10.7
Loop	9	3	5	5.6
Goto	9	3	0	4
Other	16	1	6	7.7

Basic RISC design philosophy

- Analyze the applications to find the key operations.
- Design a data path that is optimal for the key operations. (The time required to fetch the operands from their registers, run them through the ALU, and store the result back into a register, called the data path cycle time, should be made as short as possible.)
- Design instructions that perform the key operations using the data path.
- Add new instructions only if they do not slow down the machine.
- Repeat this process for other resources within the CPU, such as cache memory, memory management, floating-point coprocessors and so on.

RISC: How it turns out?

- Instructions are conceptually simple.
- Instructions are of a uniform length.
- The instructions use one (or very few) instruction format.
- The instruction set is orthogonal.
- Instructions use one (or very few) addressing mode.
- The architecture is a load-and-store architecture.
- The ISA supports two (or a few more) datatypes (typically integer and floating point).

  (The following properties are common for RISC machines, but should not be used to define a RISC architecture.)

This is fals

- Almost all instructions execute in 1 clock cycle.
- The architecture takes advantage of the strengths of software.
- The architecture should have many registers.

RISC-CISC Controversy

- Arguments of CISC proponents:
  - Richer (more complex) instruction sets improve the merit of the architecture, since operations implemented in microcode execute faster than operations implemented in software.
  - Richer instruction sets do not increase the cost of implementation (in dollars) over that of simpler instruction sets.
  - The need for upward compatibility within a family results in an increase in the instruction-set size, and upward compatibility is easier to implement in microcode.
  - Richer instruction sets simplify compiler design.
  - Complex instruction sets make cloning of a computer more difficult, thus protecting proprietary design.

RISC-CISC Controversy

- Arguments of RISC proponents:
  - The basic hardware is simpler, so it can be cheaper and *faster*, more than compensating for the increased number of instructions required to perform some operations.
  - Instruction caches easily compensate for the large number of bits in the instructions required by a RISC.
  - It is easier to compile for a RISC than for a CISC architecture.
  - Design effort, and hence development cost, for a RISC is less than for a CISC.
  - It is easier to introduce parallelism into the control unit of a RISC than a CISC.

RISC Implementation Techniques

- RISCs use pipelining to speed up instruction decoding and execution.
  - Instructions are simple and uniform and therefore pipelining is easy
- RISCs do not allow program self-modification, which eliminates the need for hardware interlocks to detect possible modifications to the program.
- RISCs use Harvard architectures; separate instruction and data streams reduce the von Neumann bottleneck.
  - Because instructions are simple, more instructions are typically needed to accomplish a task. More fetches of instructions.
- RISCs use large register sets to reduce the CPU-to-memory bandwidth and the result-register dependencies that occur with small register sets.
  - Transfer within internal registers are faster than memory access.
  - Simplified allows more silicon area for on-chip registers

RISC Implementation Techniques

- All RISCs have separate functional units for instruction processing and instruction execution, and most have independent floating-point functional units.
  - Again to avoid bottleneck at the ALU
- RISCs use delayed branches to avoid the branch penalty. The CPU always executes the instruction in the branch-delay slot; execution of branch is therefore delayed for one instruction.

- Use of delayed loads avoids the operand-fetch delay. The CPU always executes the instruction in the load-delay slot.
- RISCs prefetch branch-target instructions to reduce the branch delay.
- RISCs use specialized cache memories to decrease the memory-to-CPU delay.

RISC Implementation Techniques

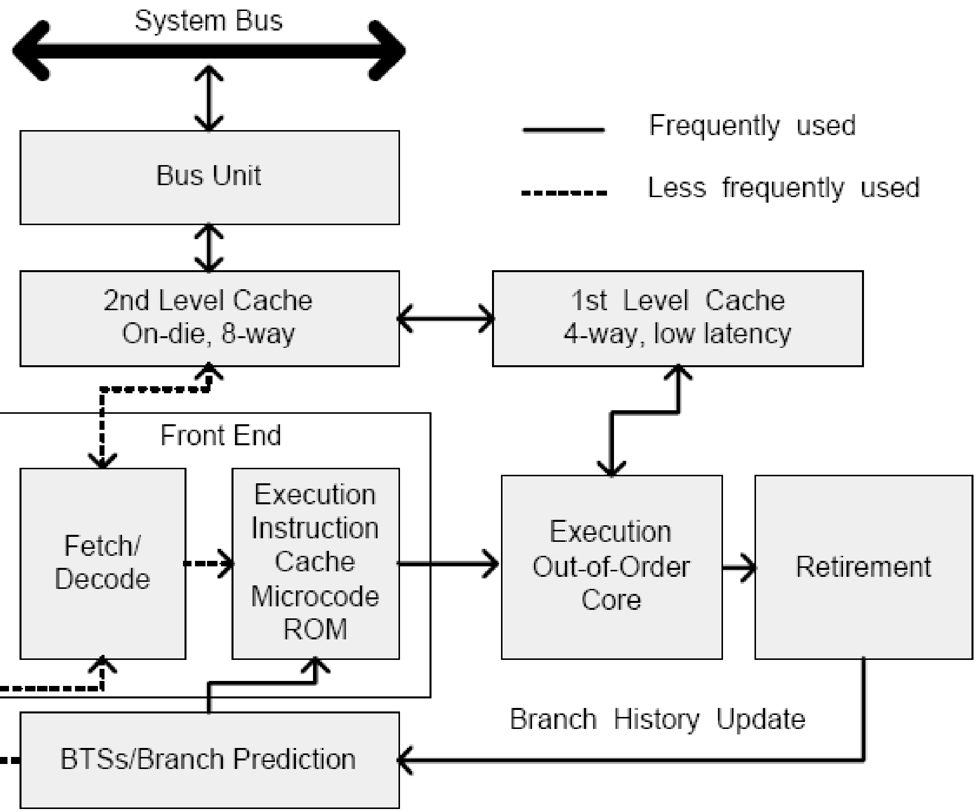
- RISCs use optimizing compilers. Optimizing compilers rearrange the code sequences to take maximum advantage of the CPU parallelism.
- Some RISCs use overlapping register sets to speed up parameter passage during subroutine calls and returns.
- Some RISCs support string operations by loading and storing multiple registers and using them as string operands.
- Some RISCs support vector operations by treating multiple registers as vector registers.

Case Study

Intel IA-32 Architecture

The P6 Family Microarchitecture

- It is a three-way superscalar, pipelined architecture.
  - The processor is able on average to decode, dispatch, and complete execution of (retire) three instructions per clock cycle.
  - To handle this level of instruction throughput, the P6 processor family uses a decoupled, 12-stage superpipeline that supports out-of-order instruction execution.
- To ensure a steady supply of instructions → 2 level cache
  - Level 1 cache provides an 8-KByte instruction cache and an 8-KByte data cache, both closely coupled to the pipeline.
  - Level 2 cache provides 256-KByte, 512-KByte, or 1-MByte static RAM that is coupled to the core processor through a full clock-speed 64-bit cache bus.



Out-of-order execution mechanism

- **Deep branch prediction**
  - allows the processor to decode instructions beyond branches to keep the instruction pipeline full.
- **Dynamic data flow analysis**
  - real-time analysis of the flow of data through the processor to determine dependencies and to detect opportunities for out-of-order instruction execution.
  - The out-of-order execution core can monitor many instructions and execute these instructions in the order that best optimizes the use of the processor's multiple execution units, while maintaining the data integrity.
- **Speculative execution**
  - execute instructions that lie beyond a conditional branch that has not yet been resolved, and ultimately to commit the results in the order of the original instruction stream.
  - To make speculative execution possible, the P6 processor microarchitecture decouples the dispatch and execution of instructions from the commitment of results.
  - The processor's out-of-order execution core uses data-flow analysis to execute all available instructions in the instruction pool and store the results in temporary registers.
  - The retirement unit then linearly searches the instruction pool for completed instructions that no longer have data dependencies with other instructions or unresolved branch predictions.
  - When completed instructions are found, the retirement unit commits the results of these instructions to memory and/or the IA-32 registers (the processor's eight general-purpose registers and eight x87 FPU data registers) in the order they were originally issued and retires the instructions from the instruction pool.

Intel NetBurst microarchitecture

- The Rapid Execution Engine
  - Arithmetic Logic Units (ALUs) run at twice the processor frequency
  - Basic integer operations can dispatch in 1/2 processor clock tick
- Hyper-Pipelined Technology
  - Deep pipeline to enable industry-leading clock rates for desktop PCs and servers
- Advanced Dynamic Execution
  - Deep, out-of-order, speculative execution engine
    - Up to 126 instructions in flight; Up to 48 loads and 24 stores in pipeline2
  - Enhanced branch prediction capability
    - Reduces the misprediction penalty associated with deeper pipelines
  - 4K-entry branch target array
- Cache subsystem
  - First level caches
    - Advanced Execution Trace Cache stores decoded instructions
    - Execution Trace Cache removes decoder latency from main execution loops
    - Execution Trace Cache integrates path of program execution flow into a single line
    - Low latency data cache
  - Second level cache
    - Full-speed, unified 8-way Level 2 on-die Advance Transfer Cache
    - Bandwidth and performance increases with processor frequency
- High-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus
  - Supports quad-pumped, scalable bus clock to achieve up to 4X effective speed
  - Capable of delivering up to 3.2 to 6.4 GBytes of bandwidth per second
- Superscalar issue to enable parallelism
- Expanded hardware registers with renaming to avoid register name space limitations
- 64-byte cache line size (transfers data up to two lines per sector)

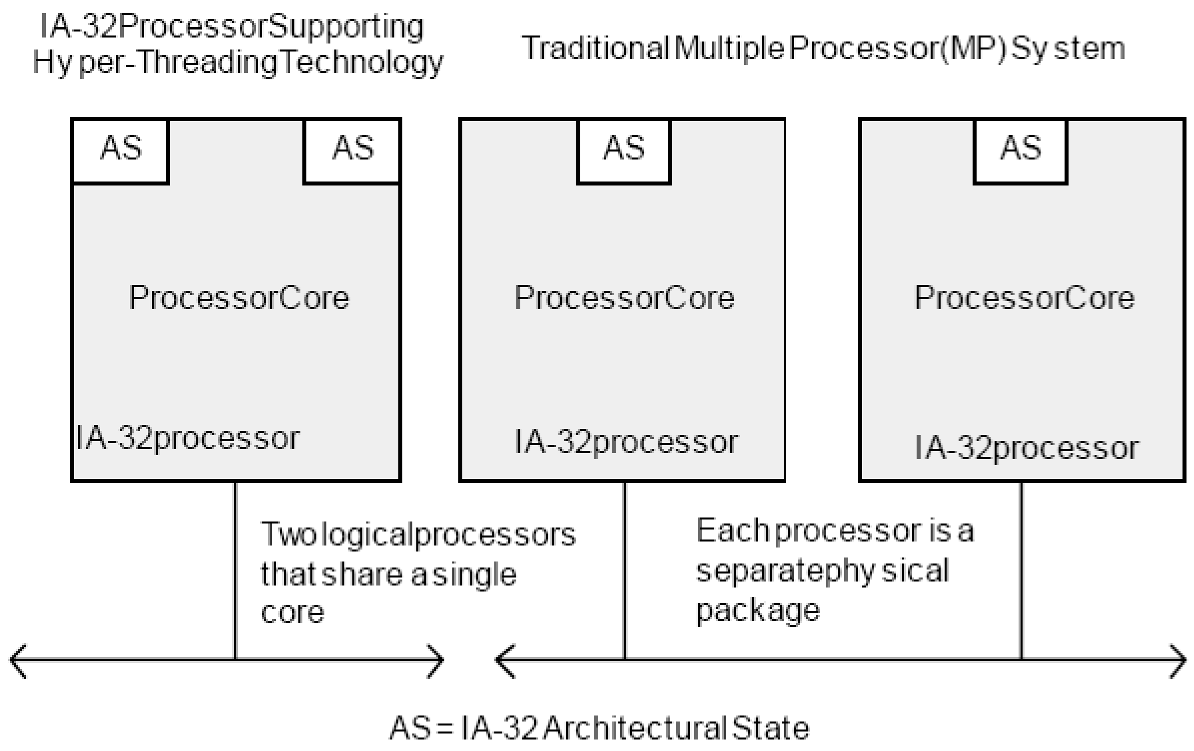
SIMD INSTRUCTIONS

- MMX technology, SSE extensions, SSE2 extensions, and SSE3 extensions.
  - group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements

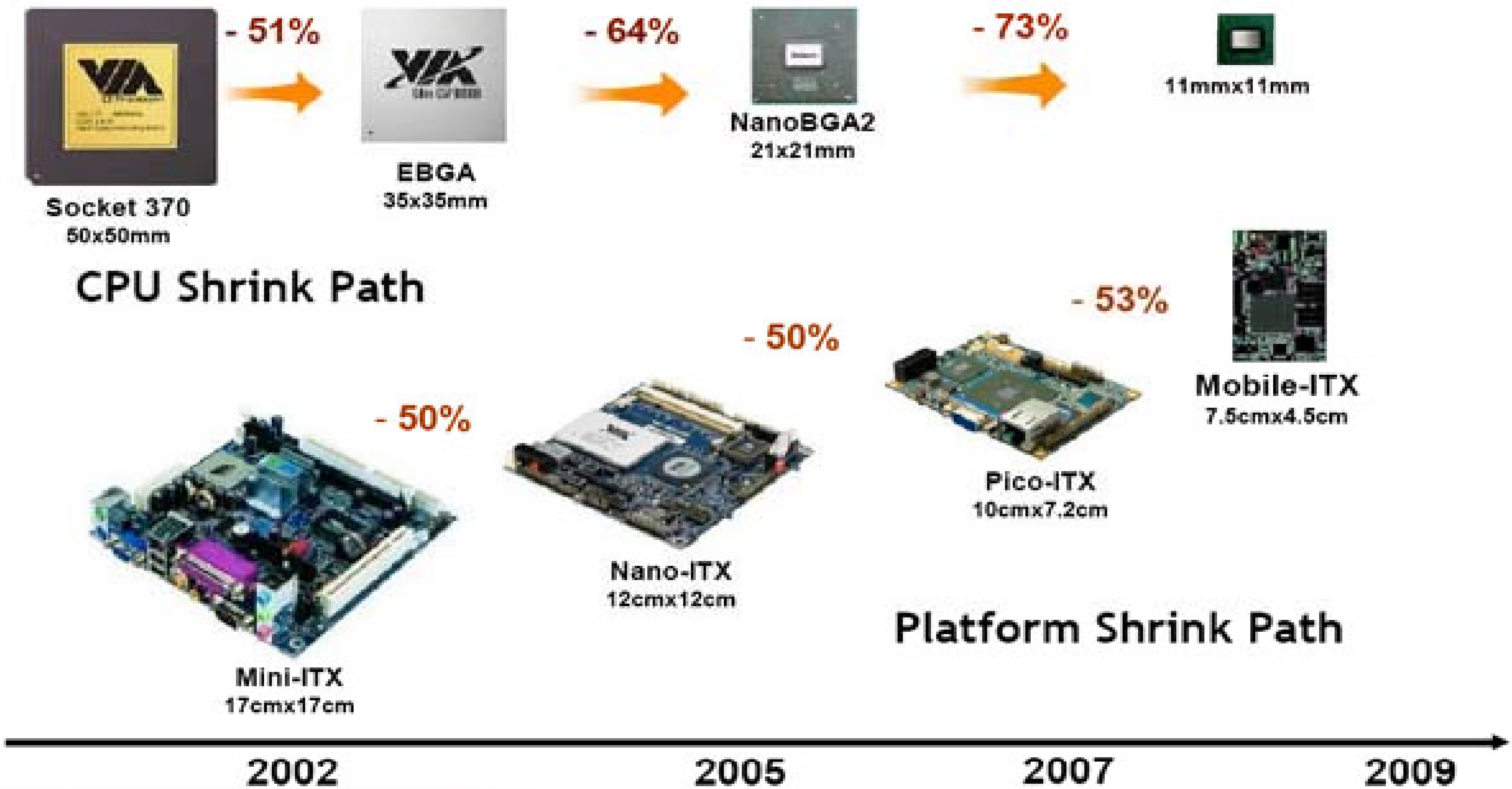
- MMX (Pentium)
  - perform SIMD operations on packed byte, word, or doubleword integers located in MMX registers.
  - useful in applications that operate on integer arrays and streams of integer data that lend themselves to SIMD processing.
- SSE extensions (Pentium III)
  - operate on packed single-precision floating-point values contained in XMM registers and on packed integers contained in MMX registers.
  - Several SSE instructions provide state management, cache control, and memory ordering operations. Other SSE instructions are targeted at applications that operate on arrays of single-precision floating-point data elements (3-D geometry, 3-D rendering, and video encoding and decoding applications).
- SSE2 extensions (Pentium 4 and Xeon)
  - operate on packed double-precision floating-point values contained in XMM registers and on packed integers contained in MMX and XMM registers.
  - SSE2 integer instructions extend IA-32 SIMD operations by adding new 128-bit SIMD integer operations and by expanding existing 64-bit SIMD integer operations to 128-bit XMM capability.
  - SSE2 instructions also provide new cache control and memory ordering operations.
- SSE3 extensions (Pentium 4 supporting Hyper-Threading Technology (built on 90 nm process).
  - SSE3 offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities.

Hyper-Threading Technology

- Enables a single physical processor to execute two or more separate code streams (threads) concurrently.
- Architecturally, an IA-32 processor that supports HT Technology consists of two or more logical processors, each of which has its own IA-32 architectural state.
  - Each logical processor consists of a full set of IA-32 data registers, segment registers, control registers, debug registers and most of the MSRs. Each also has its own advanced programmable interrupt controller (APIC).
  - share the core resources of the physical processor → includes the execution engine and the system bus interface.
    - After power up and initialization, each logical processor can be independently directed to execute a specified thread, interrupted, or halted.
- HT Technology leverages the process and thread-level parallelism found in contemporary operating systems and high-performance applications by providing two or more logical processors



Shrinking the Form Factor



Intel Processor	Date Introduced	Micro-architecture	Clock Fre-uency at Intro	Transistors	Register Sizes	System Bus Band-width	Max. Extern. Addr. Space	On-Die Caches2
Intel Pentium M Processor 755[3]	2004	Intel Pentium M Processor	2.00 GHz	140 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	4 GB	L1: 64 KB L2: 2 MB
Intel Core Duo Processor T2600[3]	2006	Improved Intel Pentium M Processor Microarchitecture; Dual Core;Intel Smart Cache,Advanced Thermal Manager	2.16 GHz	152M	GP: 32 FPU: 80 MMX: 64 XMM: 128	5.3 GB/s	4 GB	L1: 64 KB L2: 2 MB (2MB Total)
Intel Atom Processor Z5xx series	2008	Intel Atom Microarchitecture; Intel Virtualization Technology.	1.86 GHz - 800 MHz	47M	GP: 32 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	4 GB	L1: 56 KB4 L2: 512KB
64-bit Intel Xeon Processor with800 MHz System Bus	2004	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.60 GHz	125 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1;1 MB L2
64-bit Intel Xeon Processor MP with 8MB L3	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.33 GHz	675M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	5.3 GB/s1	1024 GB (1 TB)	12K µop Execution Trace Cache; 16 KB L1;1 MB L2,8 MB L3
Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.73 GHz	164 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1; 2 MB L2

Intel Pentium Processor Extreme Edition 840	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core2	3.20 GHz	230 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1;1MB L2 (2MBTotal)
Dual-Core Intel Xeon Processor 7041	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture; Dual-core3	3.00 GHz	321M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1;2MB L2 (4MBTotal)
Intel Pentium 4 Processor 672	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture;Intel Virtualization Technology.	3.80 GHz	164 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	6.4 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1;2MB L2
Intel Pentium Processor Extreme Edition 955	2006	Intel NetBurst Microarchitecture; Intel 64 Architecture; Dual Core;Intel Virtualization Technology.	3.46 GHz	376M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	12K µop Execution Trace Cache; 16 KB L1;2MB L2 (4MB Total)
Intel Core 2 Extreme Processor X6800	2006	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture;Intel Virtualization Technology.	2.93 GHz	291M	GP: 32,64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Intel Xeon Processor 5160	2006	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture;Intel Virtualization Technology.	3.00 GHz	291M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)

Intel Xeon Processor 7140	2006	Intel NetBurst Microarchitecture; Dual Core; Intel 64 Architecture;Intel Virtualization Technology.	3.40 GHz	1.3 B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	12.8 GB/s	64 GB	L1: 64 KB L2: 1MB (2MB Total) L3: 16 MB (16MB Total)
Intel Core 2 Extreme Processor QX6700	2006	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture;Intel Virtualization Technology.	2.66 GHz	582M	GP: 32,64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Quad-core Intel Xeon Processor 5355	2006	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture;Intel Virtualization Technology.	2.66 GHz	582 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	256 GB	L1: 64 KB L2: 4MB (8 MB Total)
Intel Core 2 Duo Processor E6850	2007	Intel Core Microarchitecture; Dual Core; Intel 64 Architecture;Intel Virtualization Technology;Intel Trusted Execution Technology	3.00 GHz	291 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	10.6 GB/s	64 GB	L1: 64 KB L2: 4MB (4MB Total)
Intel Xeon Processor 7350	2007	Intel Core Microarchitecture; Quad Core; Intel 64 Architecture;Intel Virtualization Technology.	2.93 GHz	582 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	1024 GB	L1: 64 KB L2: 4MB (8MB Total)
Intel Xeon Processor 5472	2007	Enhanced Intel Core Microarchitecture; Quad Core; Intel 64 Architecture;Intel Virtualization Technology.	3.00 GHz	820 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	12.8 GB/s	256 GB	L1: 64 KB L2: 6MB (12MB Total)

Intel Atom Processor	2008	Intel Atom Microarchitecture; Intel 64 Architecture; Intel Virtualization Technology.	2.0 - 1.60 GHz	47 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	Up to 64GB	L1: 56 KB4 L2: 512KB
Intel Xeon Processor 7460	2008	Enhanced Intel Core Microarchitecture; Six Cores; Intel 64 Architecture;Intel Virtualization Technology.	2.67 GHz	1.9 B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	8.5 GB/s	1024 GB	L1: 64 KB L2: 3MB (9MB Total) L3: 16MB
Intel Atom Processor 330	2008	Intel Atom Microarchitecture; Intel 64 Architecture; Dual core;Intel Virtualization Technology.	1.60 GHz	94 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	Up to 4.2 GB/s	Up to 64GB	L1: 56 KB5 L2: 512KB (1MB Total)
Intel Core i7-965 Processor Extreme Edition	2008	Intel microarchitecture code name Nehalem; Quadcore; HyperThreading Technology; Intel QPI; Intel 64 Architecture;Intel Virtualization Technology.	3.20 GHz	731 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 25 GB/s	64 GB	L1: 64 KB L2: 256KB L3: 8MB
Intel Core i7-620M Processor	2010	Intel Turbo Boost Technology, Intel Microarchitecture code name Westmere;Dualcore; HyperThreading Technology; Intel 64 Architecture;Intel Virtualization Technology.,Integrated graphics	2.66 GHz	383 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128		64 GB	L1: 64 KB L2: 256KB L3: 4MB

Intel Xeon-Processor 5680	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere;Six core;HyperThreading Technology; Intel 64 Architecture;Intel Virtualization Technology.	3.33 GHz	1.1B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; 32 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 12MB
Intel Xeon-Processor 7560	2010	Intel Turbo Boost Technology, Intel Microarchitecture code name Nehalem;Eight core;HyperThreading Technology; Intel 64 Architecture;Intel Virtualization Technology.	2.26 GHz	2.3B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 76 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 24MB
Intel Core i7-2600K Processor	2011	Intel Turbo Boost Technology, Intel Microarchitecture code name Sandy Bridge; Four core;HyperThreading Technology; Intel 64 Architecture;Intel Virtualization Technology.,Processor graphics, Quicksync Video	3.40 GHz	995M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	64 GB	L1: 64 KB L2: 256KB L3: 8MB
Intel Xeon-Processor E3-1280	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Sandy Bridge; Four core; HyperThreading Technology; Intel 64 Architecture;Intel Virtualization Technology.	3.50 GHz		GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 8MB
Intel Xeon-Processor E7-8870	2011	Intel Turbo Boost Technology, Intel Microarchitecture code name Westmere; Ten core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	2.40 GHz	2.2B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 102 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 30MB