

CHAPTER 3

PROCESSOR MICROARCHITECTURE

- INSTRUCTION SET ARCHITECTURE
- STATICALLY SCHEDULED PIPELINES
- DYNAMICALLY SCHEDULED PIPELINES
- VLIW-EPIC
- VECTOR

INSTRUCTION SET ARCHITECTURES (ISA)

- WHAT'S AN ISA?
- INSTRUCTION TYPES AND OPCODE
- INSTRUCTION OPERANDS
- EXCEPTIONS
- RISC vs. CISC
- MICROCODED IMPLEMENTATIONS
- ISA USED IN THE CLASS

INSTRUCTION SET ARCHITECTURE (ISA)

- **THE ISA IS THE INTERFACE BETWEEN SOFTWARE AND HARDWARE**

Application
Compiler/Libraries of macros and procedures
Operating system
Instruction set (ISA)
Computer architecture (organization)
Circuits (implementation of hardware functions)
Semiconductor physics

- **DESIGN OBJECTIVES**
- **FUNCTIONALITY AND FLEXIBILITY FOR OS AND COMPILERS**
- **IMPLEMENTATION EFFICIENCY IN AVAILABLE TECHNOLOGY**
- **BACKWARD COMPATIBILITY**

ISAs ARE TYPICALLY DESIGNED TO LAST THROUGH TRENDS OF CHANGES IN USAGE AND TECHNOLOGY

AS TIME GOES BY THEY TEND TO GROW

INSTRUCTION TYPES AND OPCODES

- **THE OPCODE OF AN INSTRUCTION INDICATES THE OPERATION TO PERFORM**
- **FOUR CLASSES OF INSTRUCTIONS ARE CONSIDERED:**
 - INTEGER ARITHMETIC/LOGIC INSTRUCTIONS
 - ADD, SUB, MULT
 - ADDU, SUBU, MULTU
 - OR, AND, NOR, NAND
- **FLOATING POINT INSTRUCTIONS**
 - FADD, FMUL, FDIV
 - COMPLEX ARITHMETIC
- **MEMORY TRANSFER INSTRUCTIONS**
 - LOADS AND STORES
 - TEST AND SET, AND SWAP
 - MAY APPLY TO VARIOUS OPERAND SIZES
- **CONTROL INSTRUCTIONS**
 - BRANCHES ARE CONDITIONAL
 - CONDITION MAY BE CONDITION BITS (ZCVXN)
 - CONDITION MAY TEST THE VALUE OF A REGISTER (SET BY SLT INSTRUCTION)
 - CONDITION MAY BE COMPUTED IN THE BRANCH INSTRUCTION ITSELF
 - JUMPS ARE UNCONDITIONAL WITH ABSOLUTE ADDRESS OR ADDRESS IN REGISTER
 - JAL (JUMP AND LINK) NEEDED FOR PROCEDURES

CPU OPERANDS

- **INCLUDE: ACCUMULATORS, EVALUATION STACKS, REGISTERS, AND IMMEDIATE VALUES**
- **ACCUMULATORS:**
 - ADDA <mem_address>
 - MOVA <mem_address>
- **STACK**
 - PUSH <mem_address>
 - ADD
 - POP <mem_address>
- **REGISTERS**
 - LW R1, <memory-address>
 - SW R1, <memory_address>
 - ADD R2, <memory_address>
 - ADD R1,R2,R4
 - LOAD/STORE ISAs
 - MANAGEMENT BY THE COMPILER: REGISTER SPILL/FILL
- **IMMEDIATE**
 - ADDI R1,R2,#5

MEMORY OPERANDS

- **OPERAND ALIGNEMENT**

- BYTE-ADDRESSABLE MACHINES
- OPERANDS OF SIZE S MUST BE STORED AT AN ADDRESS THAT IS MULTIPLE OF S
- BYTES ARE ALWAYS ALIGNED
- HALF WORDS (16BITS) ALIGNED AT 0, 2, 4, 6
- WORDS (32 BITS) ARE ALIGNED AT 0, 4, 8, 12, 16,...
- DOUBLE WORDS (64 BITS) ARE ALIGNED AT 0, 8, 16,...
- COMPILER IS RESPONSIBLE FOR ALIGNING OPERANDS. HARDWARE CHECKS AND TRAPS IF MISALIGNED
- OPCODE INDICATES SIZE (ALSO: TAGS IN MEMORY)

- **LITTLE vs. BIG ENDIAN**

- BIG ENDIAN: MSB IS STORED AT ADDRESS XXXXXX00
- LITTLE ENDIAN: LSB IS STORED AT ADDRESS XXXXXX00
- PORTABILITY PROBLEMS, CONFIGURABLE ENDIANNESS

ADDRESSING MODES

MODE	EXAMPLE	MEANING
REGISTER	ADD R4,R3	$\text{reg}[R4] \leftarrow \text{reg}[R4] + \text{reg}[R3]$
IMMEDIATE	ADD R4, #3	$\text{reg}[R4] \leftarrow \text{reg}[R4] + 3$
DISPLACEMENT	ADD R4, 100(R1)	$\text{reg}[R4] \leftarrow \text{reg}[R4] + \text{Mem}[100 + \text{reg}[R1]]$
REGISTER INDIRECT	ADD R4, (R1)	$\text{reg}[R4] \leftarrow \text{reg}[R4] + \text{Mem}[\text{reg}[R1]]$
INDEXED	ADD R3, (R1+R2)	$\text{reg}[R3] \leftarrow \text{reg}[R3] + \text{Mem}[\text{reg}[R1] + \text{reg}[R2]]$
DIRECT OR ABSOLUTE	ADD R1, (1001)	$\text{reg}[R1] \leftarrow \text{reg}[R1] + \text{Mem}[1001]$
MEMORY INDIRECT	ADD R1, @R3	$\text{reg}[R1] \leftarrow \text{reg}[R1] + \text{Mem}[\text{Mem}[\text{Reg}[3]]]$
POST INCREMENT	ADD R1, (R2)+	ADD R1, (R2) then $R2 \leftarrow R2 + d$
PREDECREMENT	ADD R1, -(R2)	$R2 \leftarrow R2 - d$ then ADD R1, (R2)
PC-RELATIVE	BEZ R1, 100	if $R1 = 0$, $PC \leftarrow PC + 100$
PC-RELATIVE	JUMP 200	Concatenate bits of PC and offset

ACTUAL USE OF ADDRESSING MODES

- OPTIMIZE THE COMMON CASE
- DISPLACEMENT AND IMMEDIATE ARE THE MOST COMMON ADDRESSING MODES
 - 16 BITS IS USUALLY ENOUGH FOR BOTH TYPES OF VALUES
- SEVERAL ADDRESSING MODES ARE SPECIAL CASES OF DISPLACEMENT AND IMMEDIATE
 - REGISTER INDIRECT AND MEMORY ABSOLUTE
- MORE COMPLEX ADDRESSING MODES CAN BE SYNTHESIZED
 - MEMORY INDIRECT: LW R1, @(R2)
 - LW R3, 0(R2)
 - LW R1, 0(R3)
 - POST INCREMENT: LW R1, (R2)++
 - LW R1, 0(R2)
 - ADDI R2, R2, #size
 - MORE CYCLES
 - REGISTER SPILLING

NUMBER OF MEMORY OPERANDS IN ALU OPS

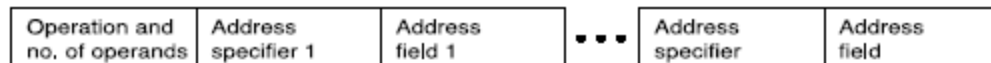
- **CONSIDER HLL STATEMENT $C \leftarrow A + B$**
- **WITH 3 MEMORY OPERANDS (MEMORY-TO-MEMORY INSTRUCTION)**
 - `ADD C,A,B`
 - LONG INSTRUCTION,
 - NO MEMORY OPERAND RE-USE
- **WITH 1 MEMORY OPERAND AND 1 REGISTER OPERAND**
 - `LW R1,A`
 - `ADD R1,B`
 - `SW R1,C`
- **WITH NO MEMORY OPERAND (LOAD/STORE ARCHITECTURE)**
 - `LW R1,A`
 - `LW R2,B`
 - `ADD R3,R1,R2`
 - `SW R3,C`

EXCEPTIONS, TRAPS AND INTERRUPTS

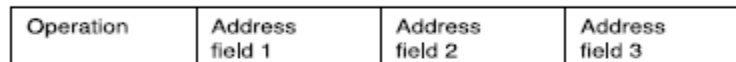
- **EXCEPTIONS ARE RARE EVENTS TRIGGERED BY THE HARDWARE AND FORCING THE PROCESSOR TO EXECUTE A HANDLER**
 - INCLUDES TRAPS AND INTERRUPTS
- **EXAMPLES:**
 - I/O DEVICE INTERRUPTS
 - OPERATING SYSTEM CALLS
 - INSTRUCTION TRACING AND BREAKPOINTS
 - INTEGER OR FLOATING-POINT ARITHMETIC EXCEPTIONS
 - PAGE FAULTS
 - MISALIGNED MEMORY ACCESSES
 - MEMORY PROTECTION VIOLATIONS
 - UNDEFINED INSTRUCTIONS
 - HARDWARE FAILURE/ALARMS
 - POWER FAILURES
- **PRECISE EXCEPTIONS:**
 - SYNCHRONIZED WITH AN INSTRUCTION
 - MUST RESUME EXECUTION AFTER HANDLER
 - SAVE THE PROCESS STATE AT THE FAULTING INSTRUCTION
 - OFTEN DIFFICULT IN ARCHITECTURES WHERE MULTIPLE INSTRUCTIONS EXECUTE

ENCODING THE ISA

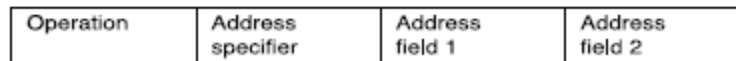
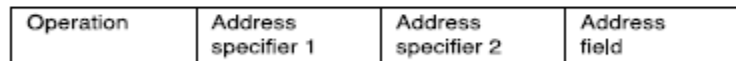
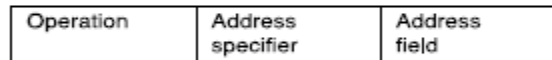
- THEORETICALLY ANY ENCODING WILL DO. HOWEVER, WATCH FOR CODE SIZE AND DECODING COMPLEXITY. DECODING IS SIMPLIFIED IF INSTRUCTION FORMAT IS HIGHLY PREDICTABLE



(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

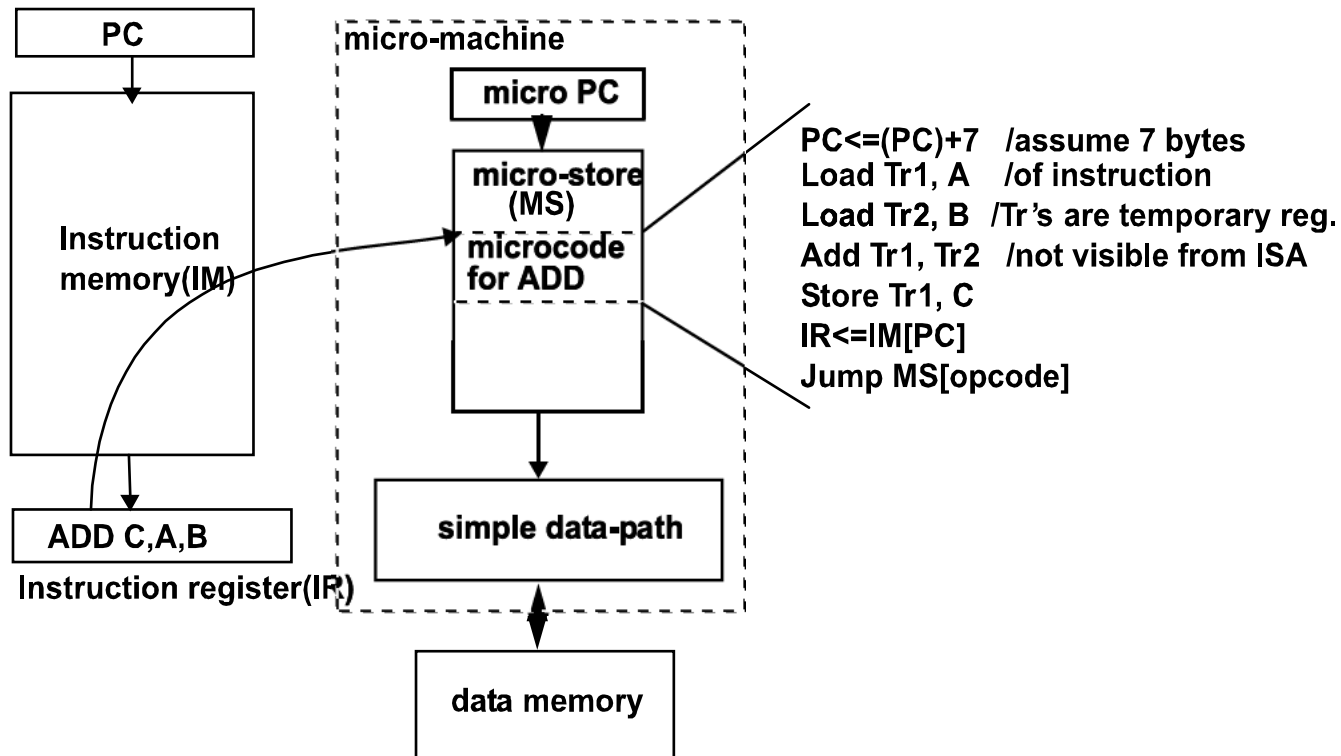


(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

INTEL iAPX 432!!

RISC vs. CISC

- COMPLEX vs REDUCED (i.e., SIMPLE) INSTRUCTION SET COMPUTERS
 - NOT DIRECTLY RELATED TO IMPLEMENTATION
 - COMPLEX RISC IMPLEMENTATIONS
 - SIMPLE CISC IMPLEMENTATIONS
- SIMPLE IMPLEMENTATION OF CISC MACHINES USE MICROCODE



- TRANSLATION OVERHEAD FROM INSTRUCTION TO MICRO INSTRUCTION

IMPORTANT ISAs AND IMPLEMENTATIONS

ISA	Company	Implementations	Type
System 370	IBM	IBM 370/3081	CISC--Legacy
x86	Intel	Intel 386, Intel Pentium IV, AMD Turion	CISC-Legacy
Motorola68000	Motorola	Motorola 68020	CISC-Legacy
Sun SPARC	Sun Microsystems	SPARC T2	RISC
PowerPC	IBM/Motorola	PowerPC-6	RISC
Alpha	DEC/Compaq/HP	Alpha 21264	RISC-Retired
MIPS	MIPS/SGI	MIPS10000	RISC
IA-64	Intel	Itanium-2	RISC

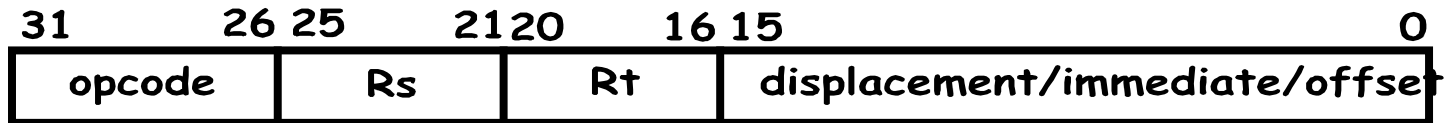
CORE ISA USED IN THE BOOK

Types	Opcode	Assembly code	Meaning	Comments
Data Transfers	LB, LH, LW, LD	LW R1,#20(R2)	$R1 \leftarrow \text{MEM}[(R2)+20]$	for bytes, half-words
	SB, SH, SW, SD	SW R1,#20(R2)	$\text{MEM}[(R2)+20] \leftarrow (R1)$	words, and double words
	L.S, L.D	L.S F0,#20(R2)	$F0 \leftarrow \text{MEM}[(R2)+20]$	single/double float load
	S.S, S.D	S.S F0,#20(R2)	$\text{MEM}[(R2)+20] \leftarrow (F0)$	single/double float store
ALU operations	ADD, SUB, ADDU, SUBU	ADD R1,R2,R3	$R1 \leftarrow (R2) + (R3)$	add/sub signed or unsigned
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1,R2,#3	$R1 \leftarrow (R2) + 3$	add/sub immediate signed or unsigned
	AND, OR, XOR,	AND R1,R2,R3	$R1 \leftarrow (R2).AND.(R3)$	bitwise logical AND, OR, XOR
	ANDI, ORI, XORI,	ANDI R1,R2,#4	$R1 \leftarrow (R2).ANDI.4$	bitwise AND, OR, XOR immediate
	SLT, SLTU	SLT R1,R2,R3	$R1 \leftarrow 1 \text{ if } R2 < R3$ else $R1 \leftarrow 0$	test on R2,R3 outcome in R1, signed or unsigned comparison
	SLTI, SLTUI	SLTI R1,R2,#4	$R1 \leftarrow 1 \text{ if } R2 < 4$ else $R1 \leftarrow 0$	test R2 outcome in R1, signed or unsigned comparison

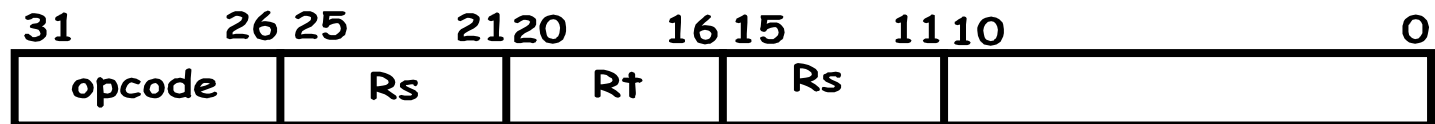
CORE ISA USED IN THE BOOK

Types	Opcode	Assembly code	Meaning	Comments
Branches/Jumps	BEQZ, BNEZ	BEQZ R1,label	PC<=label if (R1)=0	conditional branch-equal 0/not equal 0
	BEQ, BNE	BNE R1,R2,label	PC<=label if (R1)=(R2)	conditional branch-equal/not equal
	J	J target	PC<=target	target is an immediate field
	JR	JR R1	PC<=(R1)	target is in register
	JAL	JAL target	R1<=(PC)+4; PC<=target	jump to target after saving the return address in R31
Floating point	ADD.S,SUB.S,MUL.S,D IV.S	ADD.S F1,F2,F3	F1<=(F2)+(F3)	float arithmetic single precision
	ADD.D,SUB.D,MUL.D,D IV.D	ADD.D F0,F2,F4	F0<=(F2)+(F4)	float arithmetic double precision

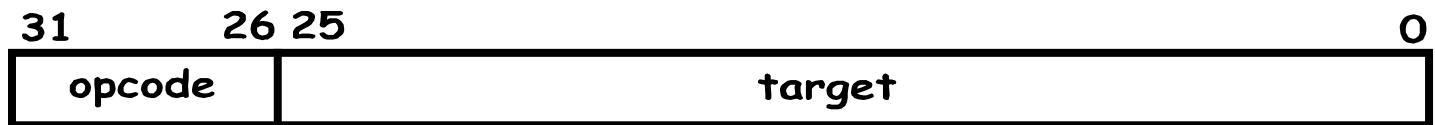
INSTRUCTION FORMATS



LW Rt, displacement(Rs)
 SW Rt, displacement(Rs)
 ADDI Rt, Rs, immediate
 BEQ Rt, Rs, offset



ADD Rd, Rt, Rs



J target
 JAL target

STATICALLY SCHEDULED PIPELINES

- 5-STAGE PIPELINE
- PIPELINES WITH OUT-OF-ORDER COMPLETION
- SUPERPIPELINED AND SUPERSCALAR CPU
- STATIC INSTRUCTION SCHEDULING
- LOOP UNROLLING
- SOFTWARE PIPELINING

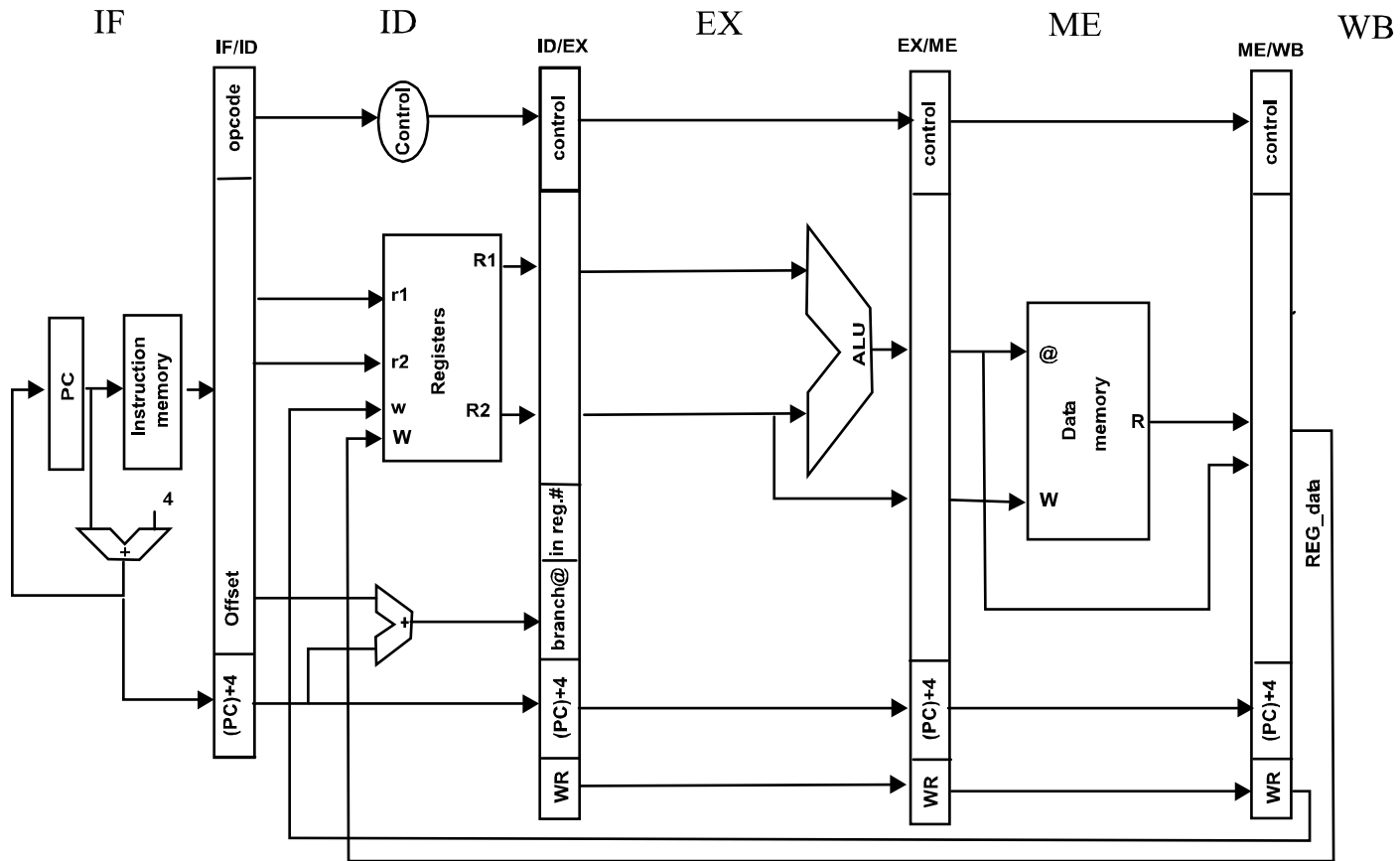
EXECUTION STEPS IN INSTRUCTIONS

Instruction	I-Fetch(IF)	I-Decode(ID)	Execute(EX)	Memory(ME)	Write-Back(WB)
LW R1,#20(R2)	Fetch; PC+=4	Decode; Fetch R2	Compute address-- (R2)+20	Read	Write in R1
SW R1,#20(R2)	Fetch; PC+=4	Decode; Fetch R1 and R2	Compute address-- (R2)+20	Write	--
ADD R1,R2,R3	Fetch; PC+=4	Decode; Fetch R2 and R3	Compute (R2)+(R3)	--	Write in R1
ADDI R1,R2,imm.	Fetch; PC+=4	Decode; Fetch R2	Compute (R2)+imm	--	Write in R1
BEQ R1,R2,offset	Fetch; PC+=4	Decode; Fetch R1 and R2 Compute target-- address (PC)+offset	Subtract (R1) and (R2) Take branch if zero	--	--
J target	Fetch; PC+=4	Decode; Take jump	--	--	--

THESE STEPS CAN BE PIPELINED

WE'LL DEAL WITH FLOATING-POINT INSTRUCTIONS LATER

5-STAGE PIPELINE



INSTRUCTIONS GO THROUGH EVERY STAGE IN PROCESS ORDER, EVEN IF THEY DON'T USE THE STAGE

- **NOTE: CONTROL IMPLEMENTATION**
 - INSTRUCTION CARRIES CONTROL
 - THIS IS A GENERAL APPROACH: "INSTRUCTION CARRIES ITS BAGGAGE"

PIPELINE HAZARDS

- **STRUCTURAL HAZARDS**

- CAUSED BY RESOURCE CONTENTION (REGISTER FILE PORT, MEMORY FETCH)
- CAN BE AVOIDED BY ADDING RESOURCES, UNLESS TOO COSTLY
- NO SUCH HAZARD IN THE 5-STAGE PIPELINE

EXAMPLE: SINGLE MEMORY IN THE 5-STAGE PIPELINE

- **DATA HAZARDS**

- DUE TO PROGRAM DEPENDENCIES (RAW, WAW, WAR)
 - BOTH FOR MEMORY AND REGISTER OPERAND ACCESSES
- DIFFERENCE BETWEEN DATA DEPENDENCIES AND DATA HAZARDS
 - SOFTWARE vs. HARDWARE IMPLEMENTATION
- IN 5-STAGE PIPELINE: ONLY RAW DEPENDENCIES ON REGISTERS CAUSE HAZARDS
 - BECAUSE ALL INSTRUCTIONS GO THROUGH EVERY PIPELINE STAGE IN PROCESS ORDER
 - IN PARTICULAR, THEY GO THROUGH THE WB STAGE IN PROCESS ORDER
 - ONLY ONE MEMORY STAGE EXECUTING MEMORY ACCESSES

- **CONTROL HAZARDS**

- BRANCH, JUMP, EXCEPTIONS

RAW HAZARDS ON REGISTER VALUES

RAW HAZARD WITH A PRECEDING ALU INSTRUCTION

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	ADD R1,R2,R3	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#20				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

- **I1 AND I5 ARE FINE**
- **BETWEEN I1 AND I4:**
 - REGISTER FORWARDING: VALUE STORED = VALUE READ
- **BETWEEN I1 AND I3**
 - VALUE IS AVAILABLE AND CAN BE FORWARDED FROM WB INPUT TO EX INPUT
- **BETWEEN I1 AND I2**
 - VALUE IS AVAILABLE AND CAN BE FORWARDED FROM ME INPUT TO EX INPUT

FORWARDING: PROVIDES DIRECT PATHS BETWEEN ME AND WB INTO EX

FORWARDING IS IMPLEMENTED BY DETECTING IN A FORWARDING UNIT (FU) THAT THE INSTRUCTION IN ME OR WB WRITES INTO ONE INPUT REGISTER OF THE INSTRUCTION IN EX

RAW HAZARDS ON REGISTER VALUES

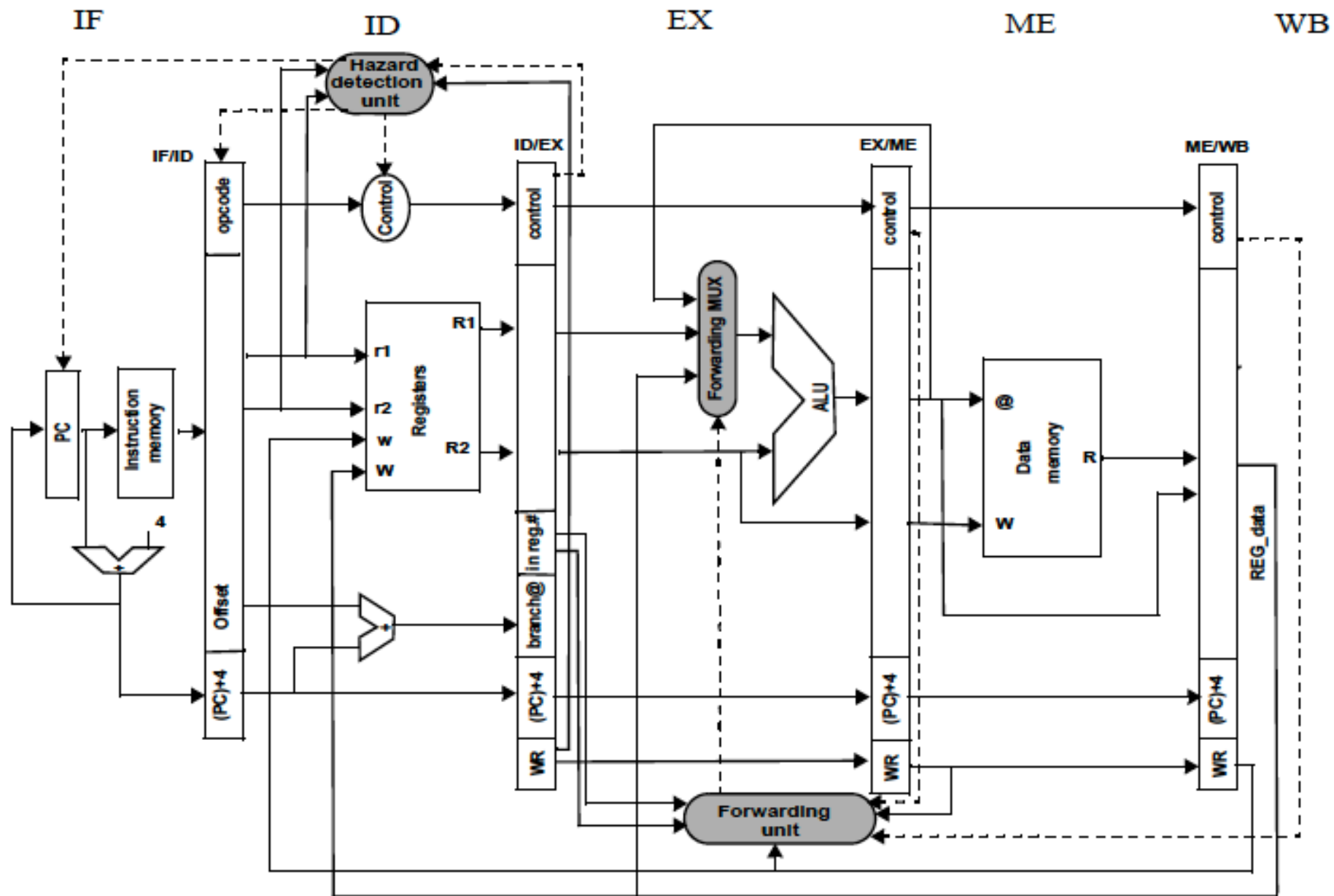
	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	EX	ME	WB			
I3	LW R5,0(R1)			IF	ID	EX	ME	WB		
I4	ORI R6,R1,#20				IF	ID	EX	ME	WB	
I5	SUBI R1,R1,R7					IF	ID	EX	ME	WB

- DEPENDENCIES WITH PRIOR LOADs POSE SPECIAL COMPLICATIONS
- IN THIS CASE IT IS NOT POSSIBLE TO FORWARD THE VALUE FROM I1 TO I2 BECAUSE THE VALUE IS AVAILABLE ONLY IN CLOCK C5
 - HOWEVER, THE VALUE CAN BE FORWARDED FROM I1 TO I3
 - I2 MUST BE STALLED TO WAIT FOR THE VALUE OF I1

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9
I1	LW R1,0(R3)	IF	ID	EX	ME	WB				
I2	ADDI R3,R1,#4		IF	ID	ID	EX	ME	WB		
I3	LW R5,0(R1)			IF	IF	ID	EX	ME	WB	
I4	ORI R6,R1,#20					IF	ID	EX	ME	WB
I5	SUBI R1,R1,R7						IF	ID	EX	ME

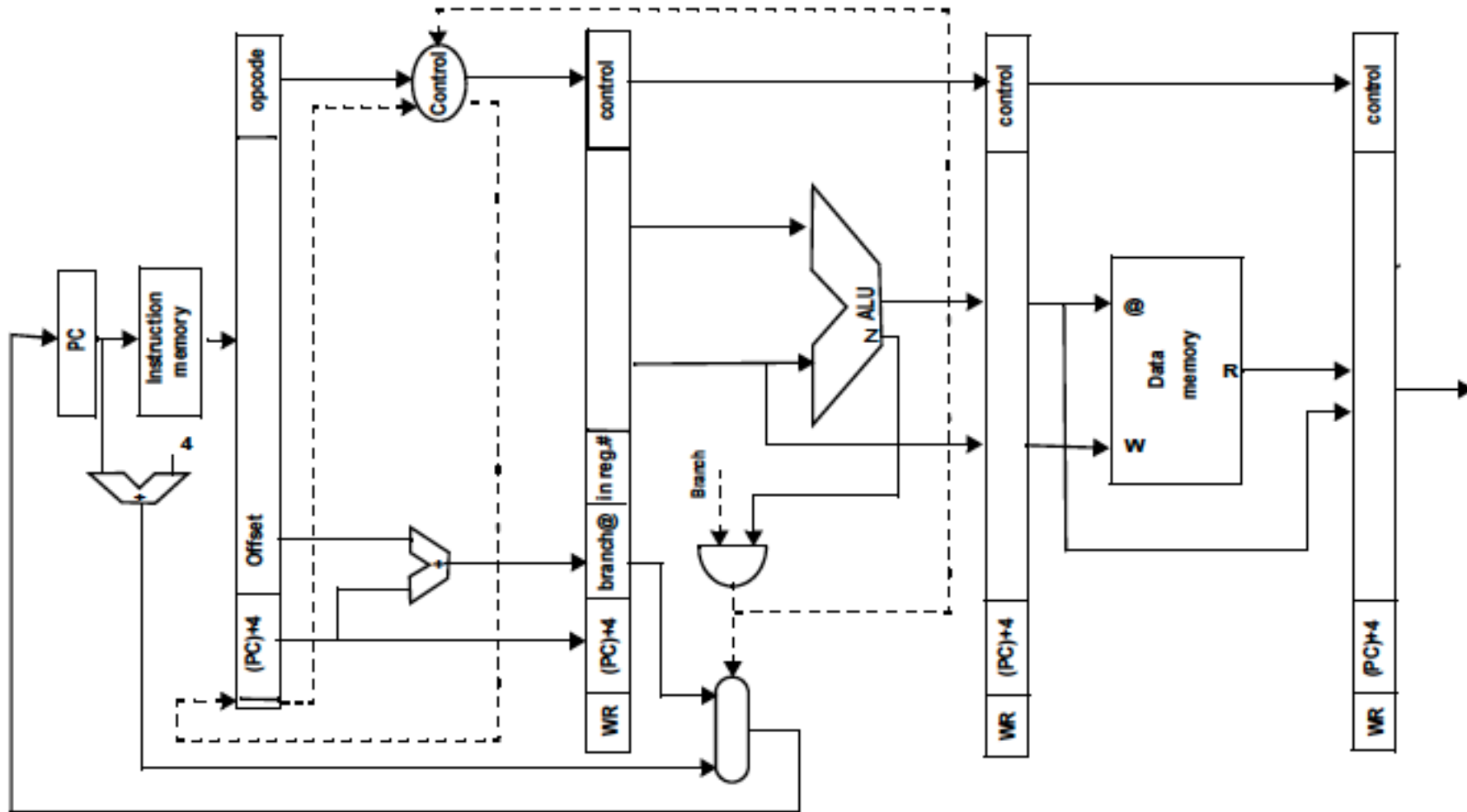
I2 AND I3 MUST BE STALLED IN IF AND ID FOR ONE CYCLE (C4) BY A HAZARD DETECTION UNIT (HDU)
IN CYCLE C4 A NOOP HAS BEEN CLOCKED IN EX (BUBBLE)

5 STAGE PIPELINE WITH FU AND HDU



CONTROL HAZARDS

- **NEED TARGET ADDRESS** (computed in ID) **AND CONDITION** (computed in EX)
 - PREDICT BRANCH NOT TAKEN AND TAKE IT IN EX IF TAKEN



IF THE BRANCH MUST BE TAKEN THEN IF AND ID MUST BE FLUSHED AND THE TARGET ADDRESS MUST BE CLOCKED INTO THE PC

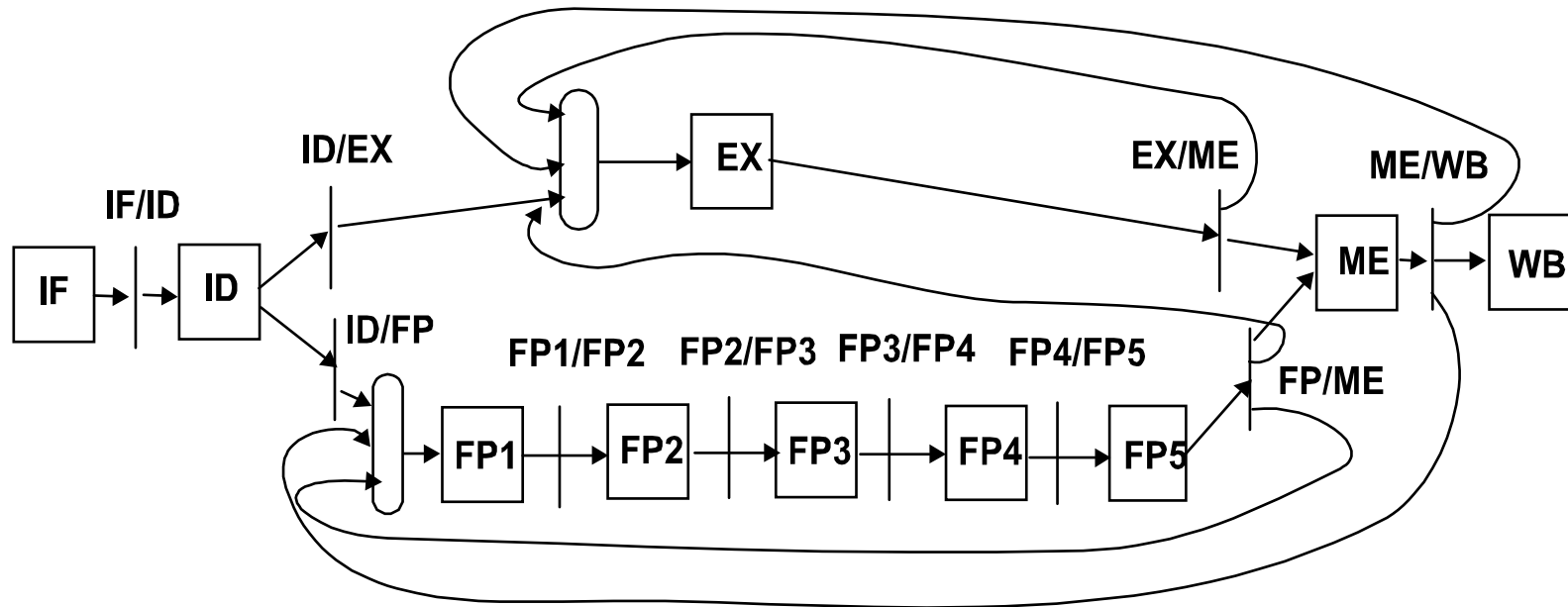
EXCEPTIONS

- **PIPELINE BENEFITS COME FROM OVERLAPPED INSTRUCTION EXECUTIONS**
 - THINGS GO WRONG WHEN THE FLOW OF INSTRUCTIONS IS SUDDENLY INTERRUPTED (EXCEPTIONS)
- **MANY EXCEPTIONS MUST BE PRECISE. ON A PRECISE EXCEPTION**
 - ALL INSTRUCTIONS PRECEDING THE FAULTING INSTRUCTION MUST COMPLETE
 - THE FAULTING INSTRUCTION AND ALL FOLLOWING INSTRUCTIONS MUST BE SQUASHED (FLUSHED)
 - THE HANDLER MUST START ITS EXECUTION
- **IT IS NOT PRACTICAL TO TAKE AN EXCEPTION IN THE CYCLE WHEN IT HAPPENS**
 - MULTIPLE EXCEPTIONS IN THE SAME CYCLE
 - IT IS COMPLEX TO TAKE EXCEPTIONS IN VARIOUS PIPELINE STAGES
 - THE MAJOR REASON IS THAT EXCEPTIONS MUST BE TAKEN IN PROCESS ORDER AND NOT IN TEMPORAL ORDER
- **INSTEAD, FLAG THE EXCEPTION AND RECORD ITS CAUSE WHEN IT HAPPENS, KEEP IT "SILENT" AND WAIT UNTIL THE INSTRUCTION REACHES THE WB STAGE TO TAKE IT.**

POSSIBLE PROBLEMS FOR PRECISE EXCEPTIONS IN CISC MACHINES

- EXCEPTIONS ARE HANDLED EASILY IN THE WB STAGE BECAUSE OUR ISA IS RISC AND MACHINE STATES ARE UPDATED AT THE END OF THE PIPELINE
- PRECISE EXCEPTIONS ARE MUCH MORE DIFFICULT IN CISC MACHINES BECAUSE OF:
 - EARLY REGISTER CHANGES DUE TO COMPLEX ADDRESSING MODES
 - AUTO-INCREMENT AND AUTO-DECREMENT ADDRESSING MODES
 - CONDITION CODES
 - PROBLEM IF THEY CAN BE SET IN MULTIPLE STAGES
 - IF SET EARLY THEY MUST BE RESTORED ON AN EXCEPTION
 - MULTICYCLE UNINSTRUCTIONS
 - WHERE TO STOP
 - HOW TO RESTART
 - USE OF REGISTERS AS WORKING STORAGE IN THE MICROCODE OF COMPLEX INSTRUCTIONS

OUT OF ORDER EXECUTION COMPLETION



- **FLOATING-POINT INSTRUCTIONS TAKE 5 CLOCKS AND ARE PIPELINED**
- **INTEGER UNIT: HANDLES INTEGER INSTRUCTIONS, BRANCHES, AND LOADs /STOREs**
- **TWO SEPARATE REGISTER FILES: INTEGER AND FP**
- **FORWARDING PATHS INTO EX AND FP1**
 - FP VALUES FORWARDED FROM FP/ME AND ME/WB TO ID/EX BECAUSE OF STOREs
- **INSTRUCTIONS WAIT IN ID UNTIL THEY CAN PROCEED DATA HAZARD FREE**
 - **STATIC SCHEDULING OF INSTRUCTIONS**
- **STILL IN-ORDER EXECUTION, BUT OUT-OF-ORDER COMPLETION**

LATENCY vs REPEAT INTERVAL

- **LATENCY OF OPERATION:**
 - MINIMUM NUMBER OF CYCLES BETWEEN AN INSTRUCTION PRODUCING A RESULT AND THE INSTRUCTION CONSUMING IT
 - DEPENDENT INSTRUCTION MUST STALL IN ID UNTIL ITS OPERAND IS FORWARDED
 - IF FUNCTIONAL UNITS ARE LINEAR PIPELINES THEN INSTRUCTION LATENCY IS THE EXECUTION TIME MINUS 1.
- **REPEAT/INITIATION INTERVAL**
 - NUMBER OF CYCLES THAT MUST ELAPSE BETWEEN CONSECUTIVE ISSUES OF INSTRUCTIONS TO THE SAME UNIT
 - IF A FUNCTIONAL UNIT IS NOT A LINEAR PIPELINE, THEN TWO CONSECUTIVE INSTRUCTIONS MAY NOT BE ISSUED TO IT IN CONSECUTIVE CYCLES BECAUSE OF STRUCTURAL HAZARDS (NOT THE CASE HERE)
 - ASSUME FOR EXAMPLE THAT THE FP UNIT IS NOT PIPELINED.
 - THEN FP INSTRUCTIONS COULD ONLY BE ISSUED EVERY 5 CLOCKS
- **FOR THE NEW FP CAPABLE MACHINE**

FUNCTIONAL UNITS	LATENCY	INITIATION INTERVAL
INTEGER ALU	0	1
LOAD	1	1
FP OP	4	1(5 IF NOT PIPELINED)

NEW STRUCTURAL HAZARDS

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
I1	ADD.S F1,F2,F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB	
I2	ADD.S F4,F2,F3		IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	L.S F10			IF	ID	EX	ME	WB			
I4	L.S F12				IF	ID	EX	ME	WB		
I5	L.S F14					IF	ID	EX	ME	WB	

- **SCAN COLUMNS FOR COMMON RESOURCE USAGE**
 - AT CYCLE C8: I1 AND I5 ARE BOTH IN THE ME STAGE
 - DOES NOT MATTER SINCE I1 DOES NOT ACCESS MEMORY
 - AT CYCLE C9: I1 AND I5 ARE BOTH IN THE WB STAGE AND BOTH STORE IN FP REGISTERS
 - STRUCTURAL HAZARD ON FP REGISTER FILE WRITE PORT
 - COULD ADD ONE WRITE PORT TO REGISTER FILE (IS THIS A GOOD IDEA???)
 - WAY TOO COMPLEX!!
 - SOLUTION: MAKE SURE THAT ONLY ONE FP OPERAND ACCESSES THE FP REGISTER FILE IN EACH CYCLE
 - STRUCTURAL HAZARD ON REGISTER FILE WRITE PORT IS SOLVED IN ID BY STALLING I5 IN I-DECODE
- **ALSO: STRUCTURAL HAZARDS ON EXECUTION UNITS**
- **TWO GENERAL TECHNIQUES TO REMOVE STRUCTURAL HAZARDS**
 - ADD HARDWARE RESOURCES
 - DEAL WITH IT BY STALLING INSTRUCTIONS TO SERIALIZE CONFLICTS

NEW DATA HAZARDS

- **WAW HAZARDS ON FP REGISTERS ARE POSSIBLE SINCE INSTRUCTIONS NOW REACH WB OUT OF ORDER**
 - ADD.D F2,F4,F6
 - L.D F2, 0(R2)
 - THE L.D MUST STALL IN ID
- **STILL NO WAR HAZARDS ON REGISTERS SINCE READS HAPPEN EARLY**
- **MORE RAW HAZARD STALLS DUE TO LONGER LATENCY INSTRUCTIONS**

	CLOCK ==>	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
I1	L.D F4, 0(R2)	IF	ID	EX	ME	WB						
I2	MULT.D F0,F4,F6		IF	ID	ID	FP1	FP2	FP3	FP4	FP5	ME	WB
I3	S.D F0, 0(R2)			IF	IF	ID	ID	ID	ID	ID	EX	ME

- **NO DATA HAZARDS ON MEMORY**
 - LOADs AND STOREs FOLLOW THE SAME PIPELINE PATH
 - ARE PROCESSED IN THREAD ORDER IN THE MEM STAGE
- **CONTROL HAZARDS ARE UNCHANGED**
 - TAKE BRANCH IN EX AND FLUSH IF AND ID

STRUCTURAL AND DATA HAZARDS

- **CHECKED IN THE ID STAGE**
- **STRUCTURAL HAZARDS:**
 - WAIT UNTIL THE FUNCTIONAL UNIT IS NOT BUSY (INITIATION INTERVAL)
 - MAKE SURE THE REGISTER WRITE PORT WILL BE AVAILABLE
- **RAW HAZARDS ON REGISTER**
 - FORWARDING
 - STALLING: HDU MUST CHECK THAT NONE OF THE SOURCE OPERANDS OF THE CURRENT INSTRUCTION IS THE DESTINATION OF ONE OF THE INSTRUCTIONS IN THE PIPELINES (CHECK WRITE REGISTER FIELD IN PIPELINE REGISTERS)
- **WAW HAZARDS**
 - HANDLE LIKE STALLS FOR RAW HAZARDS IN HDU
 - CHECK THAT THE DESTINATION REGISTER OF THE CURRENT INSTRUCTION IS NOT THE DESTINATION OF ANY OF THE INSTRUCTIONS IN THE PIPELINES
 - WAW HAZARDS ARE VERY RARE BECAUSE IT DOES NOT MAKE MUCH SENSE TO UPDATE A REGISTER TWICE WITHOUT USING THE FIRST VALUE
 - ADD.D F2,F4,F6
 - L.D F2,0(R2)
 - IF AN INSTRUCTION READING F2 IS INSERTED BETWEEN THE TWO INSTRUCTION THEN THE WAW HAZARD IS SOLVED BY THE HARDWARE SOLVING RAW HAZARDS

PRECISE EXCEPTIONS

- **CONSIDER THE FOLLOWING CODE;**

```
ADD.S F1, F2, F1
ADD R2,R1,R3
```

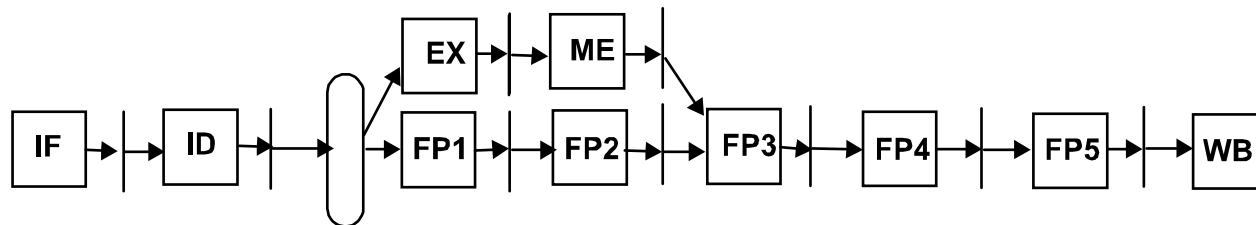
- **NO DEPENDENCY! TOTALLY INDEPENDENT**
- **HOWEVER, THERE ARE PROBLEMS WRT PRECISE EXCEPTION**
 - WHAT HAPPENS IF ADD.S CAUSES AN EXCEPTION LATE IN THE FP PIPELINE?
 - THE ADD HAS ALREADY BEEN EXECUTED

	C1	C2	C3	C4	C5	C6	C7	C8	9	10	11
ADD.S F1,F2,F1	IF	ID	FP1	FP2	FP3	FP4	FP5	ME	WB		
ADD R2,R1,R3		IF	ID	EX	ME	WB					

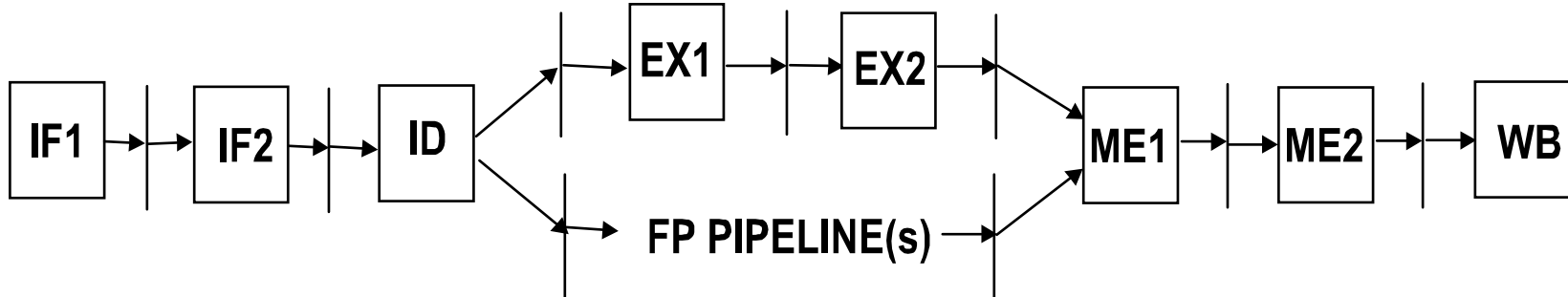
- ADD STORES ITS RESULT IN C6
- IN C7 ADD.S CAUSES AN EXCEPTION
- R2 HAS ALREADY BEEN MODIFIED

COPING WITH PRECISE EXCEPTIONS

- **FORGET PRECISION ON EXCEPTIONS**
 - NOT REALLY VIABLE TODAY IN MACHINES SUPPORTING VIRTUAL MEMORY AND IEEE FLOATING POINT STANDARD
 - GIVE STATE TO SOFTWARE AND LET THE SOFTWARE FIGURE IT OUT
 - ONLY POSSIBLE FOR SIMPLE PIPELINES
- **DEAL WITH EXCEPTIONS AS IF THEY WERE HAZARDS**
 - DO NOT ISSUE AN INSTRUCTION IN ID UNTIL IT IS SURE THAT ALL PRIOR INSTRUCTIONS ARE EXCEPTION FREE
 - DETECT EXCEPTIONS AS EARLY AS POSSIBLE IN THE EXECUTION
 - MAY STIFFLE PIPELINING
- **FORCE IN-ORDER COMPLETION**



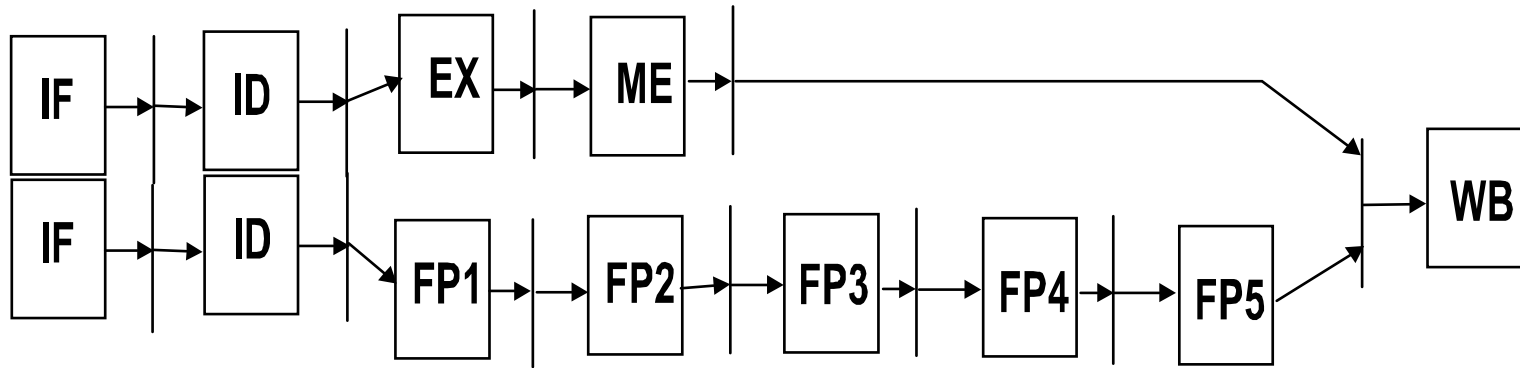
SUPERPIPELINED CPU



- **SOME STAGES IN THE 5-STAGE INTEGER PIPELINE ARE FURTHER PIPELINED**
 - TO INCREASE THE CLOCK RATE
 - HERE IF, EX, AND ME ARE NOW 2 PIPELINE STAGES
 - CLOCK IS 2X AS FAST (HOPEFULLY)
 - NOT "FREE"
 - BRANCH PENALTY WHEN TAKEN IS NOW 3 CLOCKS
 - LATENCIES IN CLOCK ARE HIGHER
- **ASSUMING UNCHANGED 5-STAGES FOR FP**

FUNCTIONAL UNITS	LATENCY	INITIATION INTERVAL
INTEGER ALU	1	1
LOAD	3	1
FP OP	4	1(5 IF NOT PIPELINED)

SUPERSCALAR CPU



- **FETCH, DECODE AND EXECUTE UP TO 2 INSTRUCTIONS PER CLOCK**
 - SAME CLOCK RATE AS BASIC PIPELINE
 - EASY BECAUSE OF THE 2 (INT AND FP) REGISTER FILES
 - ISSUE A PAIR OF INSTRUCTIONS
 - PAIR MUST BE INTEGER/BRANCH/MEMORY AND FP PAIR (COMPILER)
 - INSTRUCTIONS IN PAIR MUST BE INDEPENDENT AND HAVE NO HAZARD WITH PRIOR INSTRUCTIONS
 - SAME LATENCIES AS BASIC PIPELINE; BRANCHES EXECUTE IN EX
 - BRANCHES AND LATENCIES WASTE MORE INSTRUCTION SLOTS
 - EXCEPTIONS ARE MORE COMPLEX
- **HARD TO BUILD MORE THAN 2-WAY**
- **IPC (Instructions Per Clock) INSTEAD OF CPI ($IPC = 1/CPI$)**

ADD SUPERPIPELINING TO SUPERSCALAR

STATIC BRANCH PREDICTION

- **HARDWIRED BRANCH PREDICTION**
 - ALWAYS PREDICT UNTAKEN AND EXECUTE IN EX
 - CONDITIONAL BRANCHES AT THE BOTTOM OF A LOOP ARE MOSTLY MISPREDICTED
 - NO COMPILER ASSIST IS POSSIBLE
 - COULD PREDICT TAKEN, BUT NOT VERY USEFUL HERE (WHY?)
 - COULD BE MORE FLEXIBLE
 - BASED ON OPCODE AND/OR SIGN/SIZE OF BRANCH OPCODE
 - DECISIONS ARE MADE AT MICRO-ARCHITECTURE DESIGN TIME (BENCHMARKING)
- **COMPILE-TIME BRANCH PREDICTION**
 - EACH BRANCH INSTRUCTION HAS A "HINT" BIT SET BY THE COMPILER
 - COMPILER PROFILES THE CODE AND SETS THE HINT BIT IN BRANCH INSTRUCTIONS
 - TAKEN/NOT TAKEN PREDICTION IS MUCH MORE FLEXIBLE

STATIC INSTRUCTION SCHEDULING

- TO MAXIMIZE INSTRUCTION THROUGHPUT (IPC) OF STATIC PIPELINES
COMPILER MOVES CODE UP OR DOWN

- LOCAL (BASIC BLOCK)
- GLOBAL (ACROSS BASIC BLOCKS)
 - CYCLIC (LOOPS)
 - NON-CYCLIC (TRACES)

- WE WILL USE AN EXAMPLE

for (i=0;i<100;i++)

A[i]:= A[i] + B[i];

- AN UNOPTIMIZED CODE FOR THE LOOP IN THE BASIC PIPELINE:

Loop:	L.S F0,#4(R1)	(1)
	L.S F1,#4(R2)	(1)
	ADD.S F2,F1,F0	(2)
	S.S F2,#4(R1)	(5)
	ADDI R1,R1,#8	(1)
	ADDI R2,R2,#8	(1)
	SUBI R3,R3,#2	(1)
	BNEZ R3,Loop	(3)

TOTAL EXECUTION TIME: 15 CLOCKS; CPI = 15/8 = 1.88

LOCAL CODE SCHEDULING

- **COMPILER MOVES INSTRUCTIONS INSIDE THE LOOP BODY:**
 - MOVE SUBI UP
 - MOVE S.S DOWN

Loop:	L.S F0,#4(R1)	(1)
	L.S F1,#4(R2)	(1)
	SUBI R3,R3,#2	(1)
	ADD.S F2,F1,F0	(1)
	ADDI R1,R1,#8	(1)
	ADDI R2,R2,#8	(1)
	S.S F2,#4(R1)	(3)
	BNEZ R3,Loop	(3)

- MUST CHANGE THE DISPLACEMENT OF THE STORE (WAR HAZARD ON R1)
- **EXECUTION TIME IS 12 CLOCKS, A SPEED UP OF $15/12 = 1.25$**
- **THE PENALTY OF THE BRANCH CANNOT BE IMPROVED**
 - CHANGE THE CODE AND USE A BRANCH AT THE TOP OF THE LOOP BODY
 - DELAYED BRANCHES

**EASY BUT LIMITED BECAUSE BASIC BLOCKS ARE SMALL
USE GLOBAL SCHEDULING: LOOP UNROLLING AND SOFTWARE PIPELINING**



LOOP UNROLLING

Unroll twice	Rename FP register	Schedule	Times(basic/super)
L.S F0,0(R1)	L.S F0,0(R1)	L.S F0,0(R1)	(1) (1)
L.S F1,0(R2)	L.S F1,0(R2)	L.S F1,0(R2)	(1) (1)
ADD.S F2,F1,F0	ADD.S F2,F1,F0	L.S F3,#4(R1)	(1) (1)
S.S F2,0(R1)	S.S F2,0(R1)	L.S F4,#4(R2)	(1) (1)
L.S F0,#4(R1)	L.S F3,#4(R1)	ADD.S F2,F1,F0	(1) (2)
L.S F1,#4(R2)	L.S F4,#4(R2)	ADD.S F5,F3,F4	(1) (2)
ADD.S F2,F1,F0	ADD.S F5,F3,F4	SUBI R3,R3,#2	(1) (1)
S.S F2,#4(R1)	S.S F5,#4(R1)	ADDI R1,R1,#8	(1) (1)
ADDI R1,R1,#8	ADDI R1,R1,#8	ADDI R2,R2,#8	(1) (1)
ADDI R2,R2,#8	ADDI R2,R2,#8	S.S F2,#-8(R1)	(1) (1)
SUBI R3,R3,#2	SUBI R3,R3,#2	S.S F5,#-4(R1)	(1) (1)
BNEZ R3,Loop	BNEZ R3,Loop	BNEZ R3,Loop	(3) (4)
Copy loop twice, remove instructions, adjust displacements	Remove WAW and WAR hazards by renaming	Move loads up, move stores down, adjust displacements	

TOTAL TIME (BASIC): 14 clocks OR 7 clocks PER ITERATION OF ORIGINAL LOOP. SPEEDUP: 15/7 = 2.14

LOOP UNROLLING

- **SUPERPIPELINED PROCESSOR (2nd COLUMN)**
 - TOTAL TIME: 18 CLOCKS OR 8.5 CLOCKS PER ITERATION OF ORIGINAL LOOP
 - CLOCK RATE IS 2X AS FAST. SPEEDUP: $2 \times 15 / 8.5 = 3.52$
- **SUPERSCALAR PROCESSOR**
 - NOT SO GOOD BECAUSE THE FRACTION INT/FP INSTRUCTION FAR FROM 50-50
 - 14 CLOCKS; SPEEDUP: $15 / 7 = 2.1$

Schedule	(2nd-issue-width)	Time	Program
L.S F0,0(R1)		(1)	L.S F0,0(R1)
L.S F1,0(R2)		(1)	L.S F1,0(R2)
L.S F3,#4(R1)		(1)	L.S F3,#4(R1)
L.S F4,#4(R2)		(1)	L.S F4,#4(R2)
SUBI R3,R3,#2	ADD.S F2,F1,F0	(1)	ADD.S F2,F1,F0
ADDI R1,R1,#8	ADD.S F5,F3,F4	(1)	ADD.S F5,F3,F4
ADDI R2,R2,#8		(1)	SUBI R3,R3,#2
S.S F2,#-8(R1)		(3)	ADDI R1,R1,#8
S.S F5,#-4(R1)		(1)	ADDI R2,R2,#8
BNEZ R3,Loop		(3)	S.S F2,#-8(R1)
			S.S F5,#-4(R1)
			BNEZ R3,Loop

LOOP UNROLLING: LIMITATIONS

- WORKS WELL IF LOOP ITERATIONS ARE INDEPENDENT
- PROBLEM WITH LOOP-CARRIED RAW DEPENDENCY
 - EXAMPLE:
 for (i=5;i<100;i++)
 A[i]:= A[i-5] + B[i];
 - RECURRENCE WITH DISTANCE 5 ITERATIONS
 - UNROLL UP TO 4 TIMES
 - 5 TIMES ==> LOOP CARRIED DEPENDENCY LIMITS CODE MOTION
- CONSUMES ARCHITECTURAL REGISTERS DUE TO RENAMING
- CODE EXPANSION
 - AFFECTS I-CACHE AND MEMORY
- PROBLEM WHEN THE NUMBER OF ITERATIONS IS UNKNOWN AT COMPILE TIME

SOFTWARE PIPELINING

- ORIGINAL LOOP IS TRANSLATED INTO PIPELINED LOOP
 - PIPELINE DEPENDENT INSTRUCTIONS IN DIFFERENT ITERATIONS
- LET'S LOOK AGAIN AT OUR SIMPLE LOOP ADDING TWO VECTORS
- SINCE THE ADD.S IS THE LONG LATENCY OPERATION, WE CAN PIPELINE THE TWO LOADS AND THE ADD.S WITH THE STORE

	O_ITE1	O_ITE2	O_ITE3	O_IT4
Prologue	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---	---	---
P_ITE1	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---	---
P_ITE2	---	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0	---
P_ITE3	---	---	S.S F2,0(R1)	L.S F0,0(R1) L.S F1,0(R2) ADD.S F2,F1,F0
Epilogue				S.S F2,0(R1)

- NOTE THAT EXACTLY THE SAME OPERATIONS ARE EXECUTED IN EXACTLY THE SAME ORDER IN BOTH THE ORIGINAL LOOP AND THE PIPELINED LOOP
- THE CODE IS SIMPLY REORGANIZED



SOFTWARE PIPELINING

Prologue:	L.S F0,0(R1)
	L.S F1,0(R2)
	SUBI R3,R3,#1
	ADD.S F2,F1,F0
	ADDI R1,R1,#4
	ADDI R2,R2,#4
Loop	S.S F2,#-4(R1) (1)
	L.S F0,0(R1) (1)
	L.S F1,0(R2) (1)
	SUBI R3,R3,#1 (1)
	ADD.S F2,F1,F0 (1)
	ADDI R1,R1,#4 (1)
	ADDI R2,R2,#4 (1)
	BNEZ R3,Loop (3)
Epilogue	S.S F2,#-4(R1)

- **NO STALL, NO CODE EXPANSION, NO REGISTER RENAMING**
USE BOTH: FIRST LOOP UNROLLING THEN SOFTWARE PIPELINING

STRENGTHS AND WEAKNESSES OF STATIC PIPELINES

- **STRENGTHS**

- HARDWARE SIMPLICITY: CLOCK RATE ADVANTAGE OVER MORE COMPLEX DESIGNS
- VERY PREDICTABLE: STATIC PERFORMANCE PREDICTIONS ARE RELIABLE
- COMPILER HAS A GLOBAL VIEW OF THE CODE: e.g., OPTIMIZE LOOPS
- POWER/ENERGY ADVANTAGE

- **WEAKNESSES**

- DYNAMIC EVENTS
 - CACHE MISSES
 - CONDITIONAL BRANCHES
- NO MECHANISM TO DEAL WITH CACHE MISSES
 - FREEZE THE PROCESSOR ON A MISS
 - ONLY ONE ACCESS AT A TIME==>NO TOLERANCE FOR MISS LATENCY
- **LACK OF DYNAMIC INFORMATION**
 - MEMORY ADDRESSES==>THIS LIMITS CODE MOTION
- **NO GOOD SOLUTION FOR PRECISE EXCEPTIONS WITH OUT-OF-ORDER COMPLETION**
 - EMBEDDED VS. GENERAL PURPOSE ENVIRONMENTS
- **COMPILERS ARE COMPLEX**
 - OPTIMIZE PERFORMANCE VS. ENFORCE EXECUTION CORRECTNESS

DYNAMICALLY SCHEDULED PIPELINES

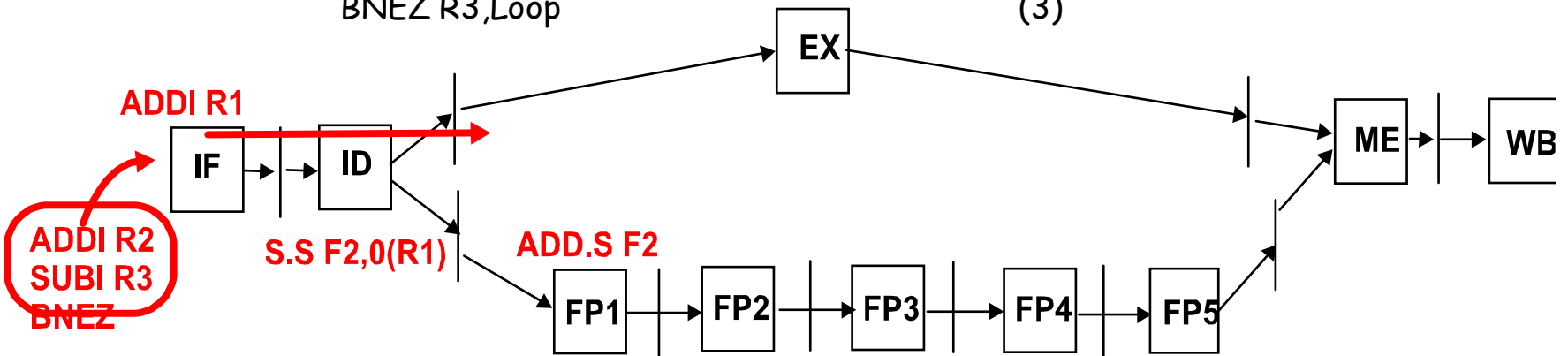
- TOMASULO ALGORITHM (SINGLE ISSUE)
- DYNAMIC BRANCH PREDICTION
- SPECULATIVE EXECUTION
- REGISTER RENAMING
- SPECULATIVE SCHEDULING
- MULTIPLE INSTRUCTION PER CLOCK
- VALUE PREDICTION
- PENTIUM III AND 4

DYNAMIC INSTRUCTION SCHEDULING

- **STATIC PIPELINES CAN ONLY EXPLOIT THE PARALLELISM EXPOSED TO IT BY THE COMPILER**
 - INSTRUCTIONS ARE STALLED IN ID UNTIL THEY ARE HAZARD-FREE AND THEN ARE SCHEDULED FOR EXECUTION
 - THE COMPILER STRIVES TO LIMIT THE NUMBER OF STALLS IN ID
 - HOWEVER COMPILER IS LIMITED IN WHAT IT CAN DO
- **POTENTIALLY THERE IS A LARGE AMOUNT OF ILP (INSTRUCTION LEVEL PARALLELISM) TO EXPLOIT (ACROSS 100s OF INSTRUCTIONS)**
 - MUST CROSS BASIC BLOCK BOUNDARIES (10s OF BRANCHES)
 - DATA-FLOW ORDER (DEPENDENCIES) --NOT THREAD ORDER
- **DYNAMIC INSTRUCTION SCHEDULING**
 - SEPARATE DECODING FROM SCHEDULING
 - DECODE INSTRUCTION THEN DISPATCH THEM IN QUEUES WHERE THEY WAIT TO BE SCHEDULED UNTIL INPUT OPERANDS ARE AVAILABLE
 - NO STALL AT DECODE FOR DATA HAZARD --ONLY STRUCTURAL HAZARDS
- **TO EXTRACT AND EXPLOIT THE VAST AMOUNT OF ILP WE MUST MEET SEVERAL CHALLENGES**
 - ALL DATA HAZARDS --RAW, WAW, AND WAR-- ARE NOW POSSIBLE BOTH ON MEMORY AND REGISTERS AND MUST BE SOLVED
 - EXECUTE BEYOND CONDITIONAL BRANCHES--SPECULATIVE EXECUTION
 - ENFORCE THE PRECISE EXCEPTION MODEL

EXAMPLE OF DYNAMIC SCHEDULING

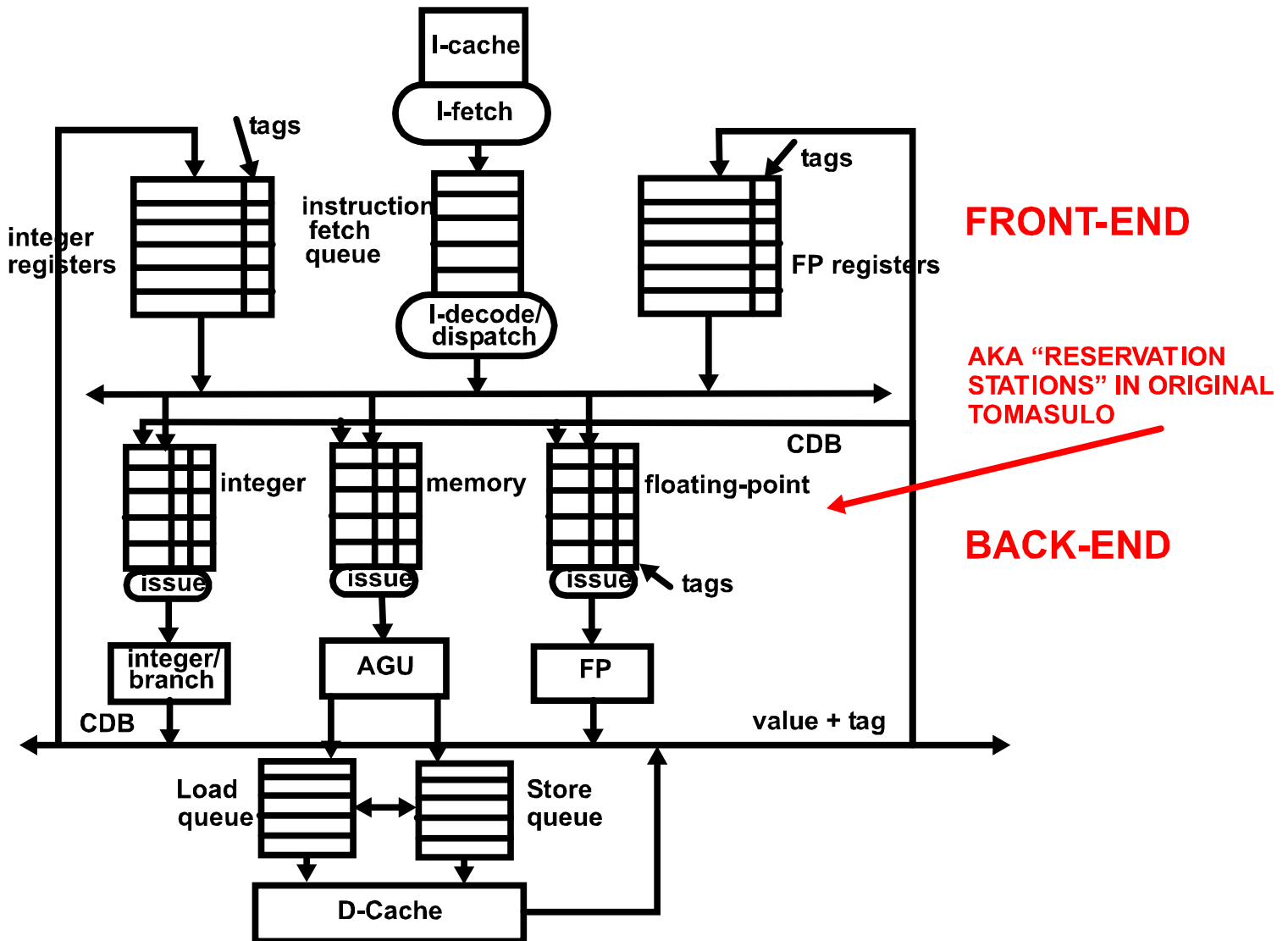
Loop	L.S F0,0(R1)	(1)
	L.S F1,0(R2)	(1)
	ADD.S F2,F1,F0	(2)
	S.S F2,0(R1)	(5)
	ADDI R1,R1,#4	(1)
	ADDI R2,R2,#4	(1)
	SUBI R3,R3,#1	(1)
	BNEZ R3,Loop	(3)



- ADDI R1 COULD PASS THROUGH ID WHILE THE STORE WAITS(WATCH FOR WAR HAZARD ON R1)
- ADDI R2 IS FETCHED AND BYPASS STORE AS WELL
- SUBI R3 IS FETCHED AND BYPASS THE STORE
- BY THAT TIME THE ADD.S IS IN FP4 AND THE STORE HAS TO STALL ONE MORE CYCLE
- COULD EVEN HAVE THE BRANCH BYPASS THE STORE (!!!!!)

LIMITS TO THIS APPROACH. DO SOMETHING NEW!!!

TOMASULO ALGORITHM



TOMASULO

- **FRONT-END**

- INSTRUCTIONS ARE FETCHED
- THEY ARE STORED IN A FIFO QUEUE CALLED "INSTRUCTION FETCH QUEUE" (IFQ)
- WHEN AN INSTRUCTION REACHES THE TOP OF THE IFQ IT IS
 - DECODED AND
 - DISPATCHED TO AN ISSUE QUEUE (INTEGER/BRANCH, MEMORY, OR FLOATING-POINT)
 - EVEN IF SOME OF ITS INPUT OPERANDS ARE NOT READY (being computed)

- **BACK-END**

- INSTRUCTIONS IN ISSUE QUEUES WAIT FOR THEIR INPUT (REGISTER) OPERANDS
- ONCE REGISTER OPERANDS ARE READY INSTRUCTIONS CAN BE SCHEDULED FOR EXECUTION, PROVIDED THEY WILL NOT CONFLICT FOR THE CDB (COMMON DATA BUS) OR THEIR FUNCTIONAL UNIT
- INSTRUCTIONS EXECUTE IN THEIR FUNCTIONAL UNIT AND THEIR RESULT IS PUT ON THE CDB
- ALL INSTRUCTIONS IN QUEUES AND ALL REGISTERS IN BOTH REGISTER FILES "WATCH" THE CDB AND GRAB THE VALUE THEY ARE WAITING FOR.

TOMASULO: HAZARDS ON REGISTERS

- **AT DISPATCH, EACH INPUT REGISTER OPERAND (+TAG) IS FETCHED FROM REGISTERS**
 - IF THE TAG IS NOT VALID
 - THE VALUE IS NOT PENDING IN THE BACK-END
 - THE REGISTER VALUE IS VALID AND IS SENT TO THE QUEUE (OPERAND READY)
 - IF THE TAG IS VALID
 - THE VALUE IS PENDING IN THE BACK-END
 - THE REGISTER VALUE IS STALE AND THE TAG IS SENT TO Q INSTEAD (OPERAND NOT READY)
- **MOREOVER THE OUTPUT REGISTER OPERAND IS ASSIGNED A TAG (THE ISSUE Q ENTRY NUMBER WHERE INSTRUCTION IS DISPATCHED)**
 - THE TAG IS STORED IN THE REGISTER FILE AND IS RECLAIMED WHEN THE INSTRUCTION HAS WRITTEN ITS VALUE ON THE CDB AND RELEASES ITS Q ENTRY
 - THE TAG OF THE RESULT IS CARRIED BY THE CDB AND "SNOOPED" BY QUEUES AND BY REGISTER FILES
 - WHEN TAG MATCH IS DETECTED THE VALUE IS STORED IN Q OR IN REGISTER
 - TAG IS INVALIDATED IN REGISTER
 - OPERAND IN INSTRUCTION IS VALID

**THIS IS A FORM OF DYNAMIC REGISTER RENAMING
REGISTER VALUES ARE RENAMED TO Q ENTRY NUMBER
MULTIPLE VALUES FOR THE SAME REGISTER MAY BE PENDING AT ANY TIME**

TOMASULO: MEMORY HAZARDS

- ALL TYPES OF HAZARDS ARE POSSIBLE ON MEMORY (RAW, WAR, WAW)
- LOAD/STORE Q: STAGING BUFFER TO SOLVE MEMORY HAZARDS
- STORES ARE SPLIT IN 2 SUB-INSTRUCTIONS
 - ONE COMPUTES THE ADDRESS, THE OTHER WAITS FOR DATA
 - BOTH ARE DISPATCHED TO THE MEMORY AND RESULTS ARE LATCHED IN THE L/S Q
 - L/S RESOLVES MEMORY HAZARDS (MEMORY DISAMBIGUATION)
- **ISSUE TO CACHE FROM MEMORY Q**
 - LOADS CAN ISSUE TO AGU AND L/S Q WHEN THEIR ADDRESS IS READY
 - BOTH STORE SUB-INSTRUCTIONS ISSUE TO AGU AND L/S Q
 - MEMORY HAZARDS (RAW, WAR, WAW) ARE RESOLVED IN THE LOAD/STORE Q
 - LOAD/STORE Q KEEPS TRACK OF THE DISPATCH ORDER OF LOADS AND STORES
 - L/S Q ENTRIES ARE RESERVED AT DISPATCH
 - LOAD CAN ISSUE TO CACHE IF NO STORE WITH SAME ADDRESS IS BEFORE IT
 - STORE CAN ISSUE TO CACHE IF NO STORE OR LOAD WITH SAME ADDRESS IS BEFORE IT
 - OTHERWISE ACCESS WAITS IN L/S Q
 - IF AN ADDRESS IS UNKNOWN, IT IS ASSUMED TO BE THE SAME.
 - WORST CASE, TO ENFORCE CORRECTNESS
 - THIS IS A VERY CONSERVATIVE APPROACH

TOMASULO: STRUCTURAL AND CONTROL HAZARDS

- **STRUCTURAL:**
 - I-FETCH MUST STALL IF THE IFQ IS FULL
 - DISPATCH MUST STALL IF ALL ENTRIES IN THE ISSUE Q OR L/S Q ARE OCCUPIED
 - INSTRUCTIONS CANNOT BE ISSUED IN CASE OF CONFLICTS FOR THE CDB OR FU
- **CONTROL(CONDITIONAL BRANCHES):**
 - DISPATCHER STALLS WHEN IT REACHES A BRANCH INSTRUCTION
 - BRANCHES ARE DISPATCHED TO INTEGER ISSUE Q
 - BRANCHES ARE TREATED AS INTEGER INSTRUCTIONS
 - THEY WAIT FOR THEIR REGISTER OPERANDS
 - THEY PUT THEIR OUTCOME ON THE CDB
 - IF UNTAKEN, THEN DISPATCH RESUMES FROM THE IFQ
 - IF TAKEN, THEN DISPATCH CLEARS THE IFQ AND DIRECTS I-FETCH TO FETCH THE TARGET I-STREAM
 - TO SPEED THINGS UP, THE FRONT-END COULD PREFETCH AND PREDECODE SOME INSTRUCTIONS IN THE TARGET I-STREAM WHILE THE BRANCH IS IN THE BACK-END.
- **CONTROL(PRECISE EXCEPTIONS)**
 - NOT SUPPORTED

DISCUSSION

- **WAW AND WAR DEPENDENCIES ARE ALSO CALLED "FALSE" OR "NAME" DEPENDENCIES**
 - RAW DEPENDENCIES ARE CALLED "TRUE" DEPENDENCIES
 - FALSE OR NAME DEPENDENCIES ARE DUE TO LIMITED MEMORY RESOURCES
- **TOMASULO ALGORITHM SOLVES WAW AND WAR HAZARDS DUE TO FALSE DEPENDENCIES ON REGISTER OPERANDS BY DISPATCHING INSTRUCTIONS IN ORDER AND RENAMING REGISTERS TO ISSUE Q ENTRY NUMBERS**
- **EXAMPLE:**

```
I1  L.S F0,0(R1)
I2  ADD.S F1,F1,F0
I3  L.S F0, 0(R2)
```

- I3 MAY COMPLETE ITS EXECUTION BEFORE I1, IF I1 MISSES AND I3 HITS IN CACHE
- HOWEVER I2 WAITS ON THE TAG OF I1, NOT ON F0 OR ON THE TAG OF I3. THUS I2 WAITS FOR THE VALUE OF F0 FROM I1 (WAR HAZARD ON F0 IS SOLVED)
- THE TAG OF F0 IN THE REGISTER FILE IS SET TO I1'S TAG WHEN I1 IS DISPATCHED
- THEN IT IS SET TO THE TAG OF I3 WHEN I3 IS DISPATCHED
- EVEN IF I3 COMPLETES WAY BEFORE I1, THE FINAL VALUE OF F0 WILL BE I3'S (WAW HAZARD ON F0 IS SOLVED)
- THE VALUE OF F0 PRODUCED BY I1 IS NEVER STORED IN REGISTER. IT IS A FLEETING VALUE, ONLY CONSUMED BY I2.

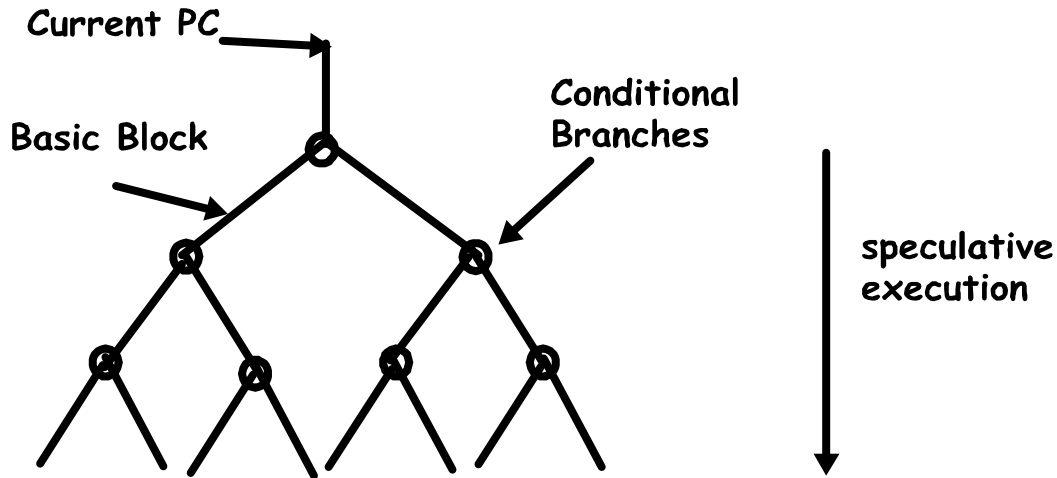
EXAMPLE

		Dispatch	Issue	Exec/ start	Exec/ complete	Cache	CDB	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	15	(16)	--	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	wait for R3
I10	L.S F0,0(R1)	15	16	(17)	17	(18)	(19)	wait for I9 (in dispatch)
I11	L.S F1,0(R2)	16	17	(18)	18	(19)	(20)	
I12	ADD.S F2,F1,F0	17	21	(22)	26	--	(27)	wait for F1

- EACH ENTRY IS CLOCK CYCLE NUMBER
- FILL TABLE CLOCK BY CLOCK
- INSTRUCTIONS ARE ISSUED AND START EXECUTION OUT OF ORDER
- LARGE OVERHEAD TO MANAGE INSTRUCTIONS.
 - LATENCY OF OPERATION IS EFFECTIVELY INCREASED
- STRUCTURAL HAZARDS: CDB/FU CONFLICTS
- COULD TAKE ADVANTAGE OF STATIC SCHEDULING (E.G., SUBI-->BRANCH)
- BRANCH ACTS AS A BARRIER TO PARALLELISM

EXECUTION BEYOND UNRESOLVED BRANCHES

- **BASIC BLOCK: BLOCK OF CONSECUTIVE INSTRUCTIONS WITH NO BRANCH AND NO TARGET OF BRANCH**

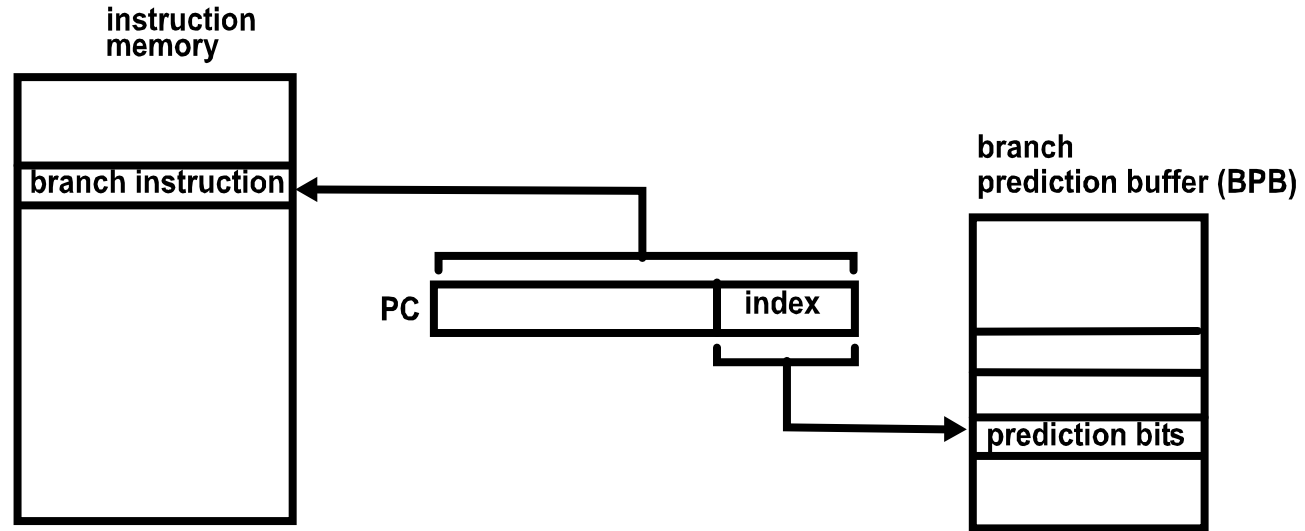


» LOOKS LIKE A TREE OF POSSIBILITIES

- **ALL-PATH EXECUTION: EXECUTE ALL PATHS AFTER BRANCH AND THEN CANCEL ALL BUT ONE PATH**
 - VERY HARDWARE INTENSIVE
 - HARD TO KEEP TRACK OF ORDER OF INSTRUCTIONS IN A TREE
 - UNWANTED EXCEPTIONS
- **PREDICT BRANCHES AND EXECUTE MOST LIKELY PATH**
 - ROLL BACK MECHANISM IN CASE PREDICTION IS WRONG
- **INTERMEDIATE SOLUTION: MULTI-PATH EXECUTION**
 - FOLLOW MOST LIKELY PATH; FOLLOW BOTH PATHS IF BRANCH IS NOT PREDICTABLE

DYNAMIC BRANCH PREDICTION

- **BRANCH PREDICTION BUFFER (BPB) ACCESSED WITH INSTRUCTION IN I-FETCH**



- **BRANCH PREDICTION BUFFER (BPB)**
 - SMALL MEMORY INDEXED WITH LSBs OF PC IN I-FETCH
 - PREDICTION IS DROPPED IF NOT A BRANCH
 - OTHERWISE THE PREDICTION BITS ARE DECODED INTO T/NT PREDICTION
 - ONCE THE BRANCH CONDITION IS KNOWN AND IF IT IS INCORRECT ROLLBACK EXECUTION
 - UPDATE PREDICTION BITS
 - ALIASING IN BPB (DIFFERENT BRANCHES AFFECT EACH OTHERS' PREDICTIONS)

1-BIT PREDICTOR

- **EACH BPB ENTRY IS 1 BIT**
 - BIT RECORDS THE LAST OUTCOME OF THE BRANCH
 - PREDICTS THAT NEXT OUTCOME IS SAME AS LAST OUTCOME
- **IN THE CONTEXT OF LOOPS:**
Loop 1: ---

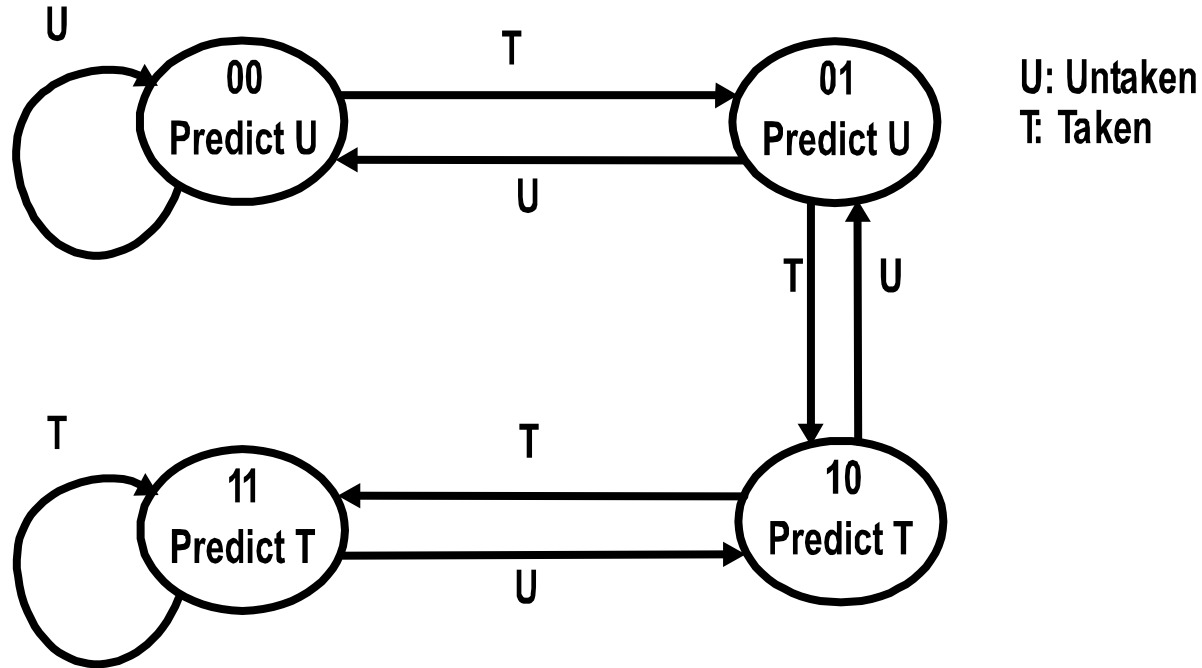
Loop2: ---

 BEZ R2, Loop2

 BNEZ R3, Loop1
- **BEZ IS ALWAYS MISPREDICT TWICE FOR EVERY LOOP EXECUTION**
 - ONCE ON ENTRY AND ONCE ON EXIT
 - THE MISPREDICT ON EXIT IS UNAVOIDABLE (DON'T KNOW WHEN LOOP ENDS)
 - BUT THE NEXT MISPREDICT ON ENTRY COULD BE AVOIDED (ON EXIT IT IS KNOWN THAT THE NEXT OUTCOME WILL BE "TAKEN")
- **SOLUTION : USE A 2-BIT PREDICTOR**

TWO-BIT PREDICTOR

- 2-BIT UP-DOWN SATURATING COUNTER IN EACH ENTRY OF THE BPB



- TAKEN==> ADD 1; UNTAKEN: SUBTRACT 1**
 - NOW IT TAKES 2 MISPREDICTIONS IN A ROW TO CHANGE THE PREDICTION
 - FOR THE NESTED LOOP, THE MISPRECTION AT ENTRY IS AVOIDED
- COULD HAVE MORE THAN 2-BITS, BUT TWO BITS COVER MOST PATTERNS (LOOPS)**

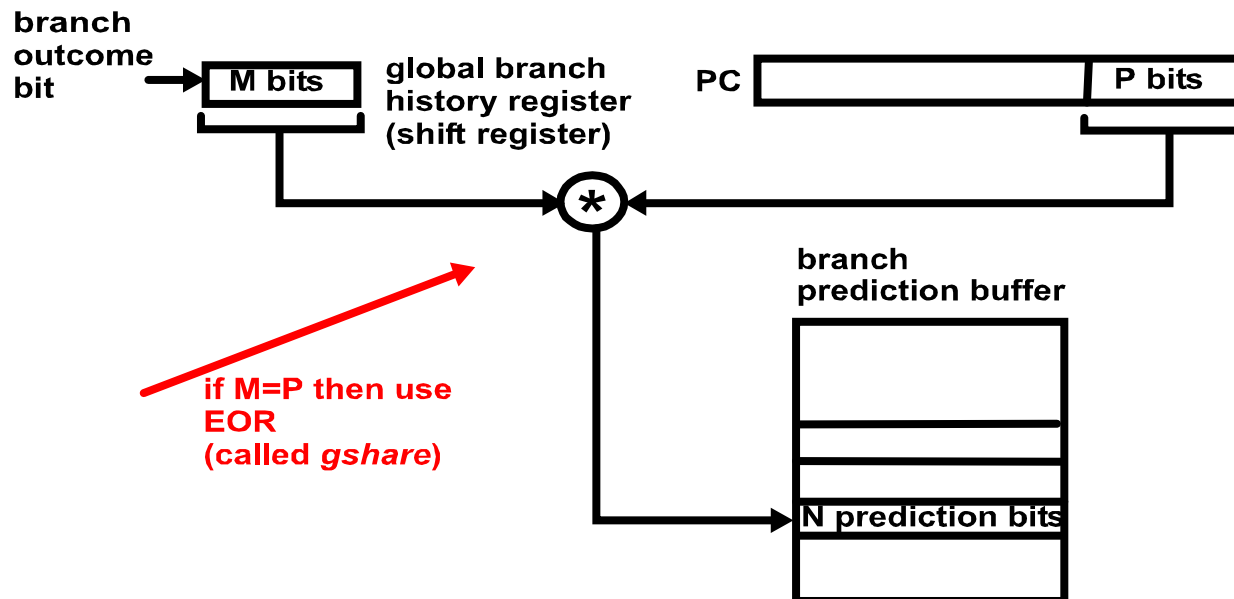
CORRELATING BRANCH PREDICTORS

- **TO IMPROVE ON TWO-BIT PREDICTORS, WE NEED TO LOOK AT OTHER BRANCHES THAN BRANCHES IN LOOPS**
 - CONSIDER THE FOLLOWING CODE SNIPPET:

if (a==2) then a:=0;
if (b==2) then b:=0;
if (a!=b) then ---
 - IF THE FIRST TWO CONDITIONS SUCCEED, THEN THE 3RD WILL FAIL
 - IN OTHER WORDS, THE 3RD BRANCH IS CORRELATED WITH THE FIRST 2
 - PREVIOUS PREDICTORS CAN'T GET THIS BECAUSE THEY KEEP TRACK OF THE CURRENT BRANCH HISTORY ONLY
- **IN GENERAL A BRANCH MAY BEHAVE DIFFERENTLY IF IT IS REACHED THROUGH DIFFERENT CODE SEQUENCES**
 - THE CODE SEQUENCE CAN BE CHARACTERIZED BY THE OUTCOME OF THE LATEST BRANCHES TO EXECUTE
- **WE CAN USE N BITS OF PREDICTION AND THE OUTCOMES OF THE LAST M BRANCHES TO EXECUTE**
 - GLOBAL vs. LOCAL HISTORY
 - NOTE THAT THE BRANCH ITSELF MAY BE PART OF THE GLOBAL HISTORY

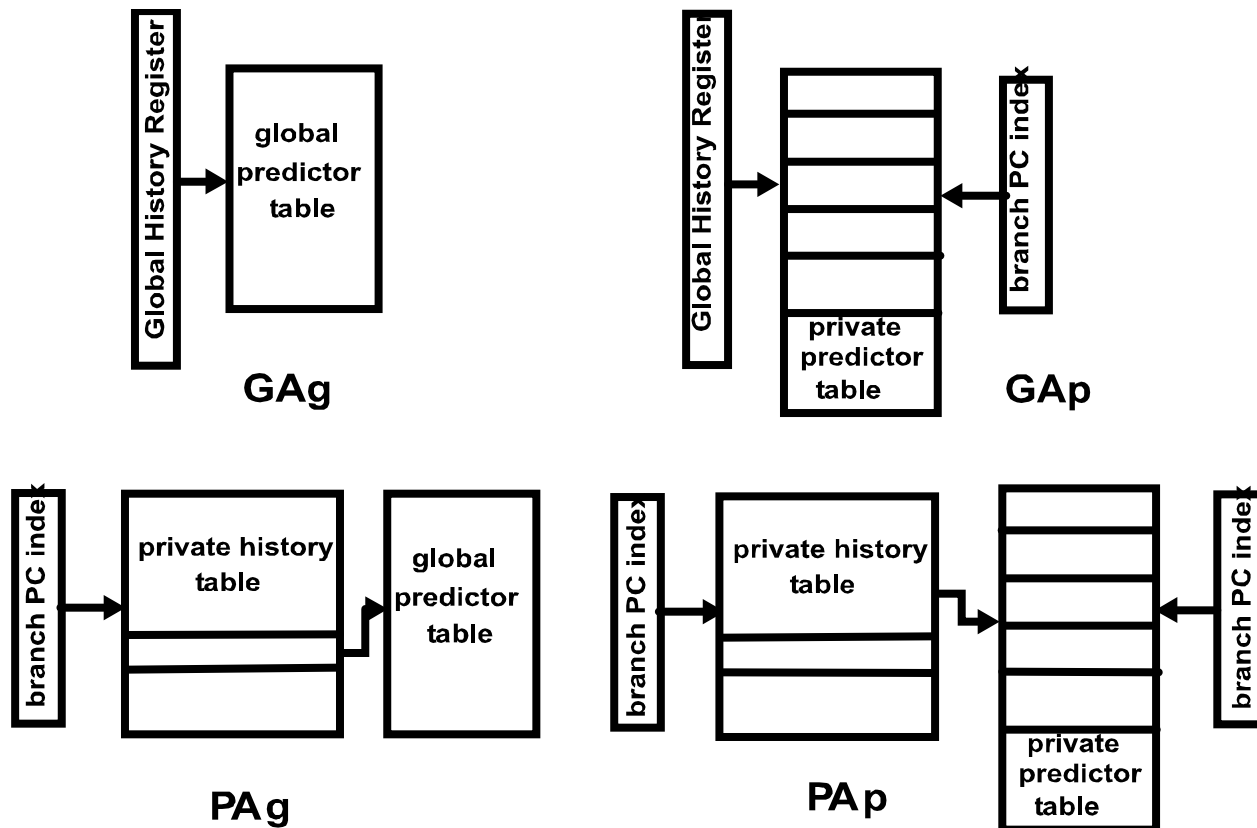
(M,N) BPB

- OUTCOMES OF THE LAST M BRANCHES IS THE GLOBAL BRANCH HISTORY
- USE N BIT PREDICTOR
- BPB IS INDEXED WITH P BITS OF THE BRANCH PC
- BPB SIZE: $N \times 2^M \times 2^P$



- IN THIS PREDICTOR WE USE GLOBAL HISTORY TO DIFFERENTIATE BETWEEN VARIOUS BEHAVIORS OF A PARTICULAR BRANCH
- THIS CAN BE GENERALIZED: TWO LEVEL PREDICTORS

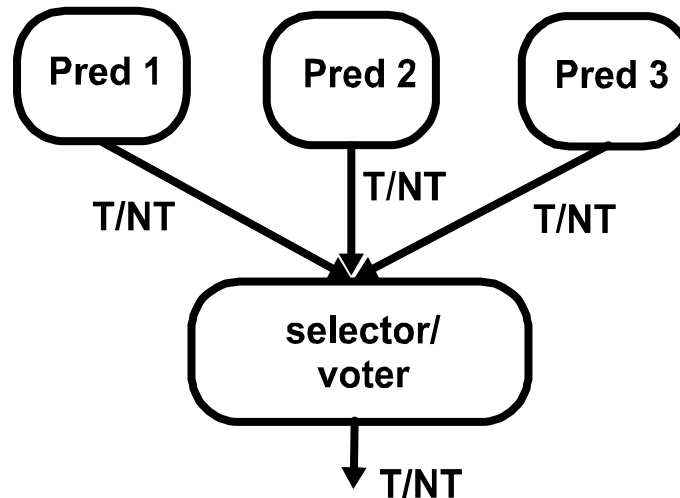
TWO-LEVEL PREDICTORS



- IN A 2-LEVEL PREDICTOR WE ADD HISTORY BITS TO ACCESS THE BTB
 - HISTORY CAN BE GLOBAL (ALL BRANCHES) OR PRIVATE (THIS BRANCH ONLY)
 - NOTATION: THE FIRST (CAPITAL) LETTER REFERS TO THE TYPE OF HISTORY; THE LAST LETTER REFERS TO WHETHER EACH PREDICTOR IS PRIVATE IN THE TABLE.
 - G OR g MEANS "GLOBAL"; P OR p MEANS "PRIVATE"

COMBINING PREDICTORS

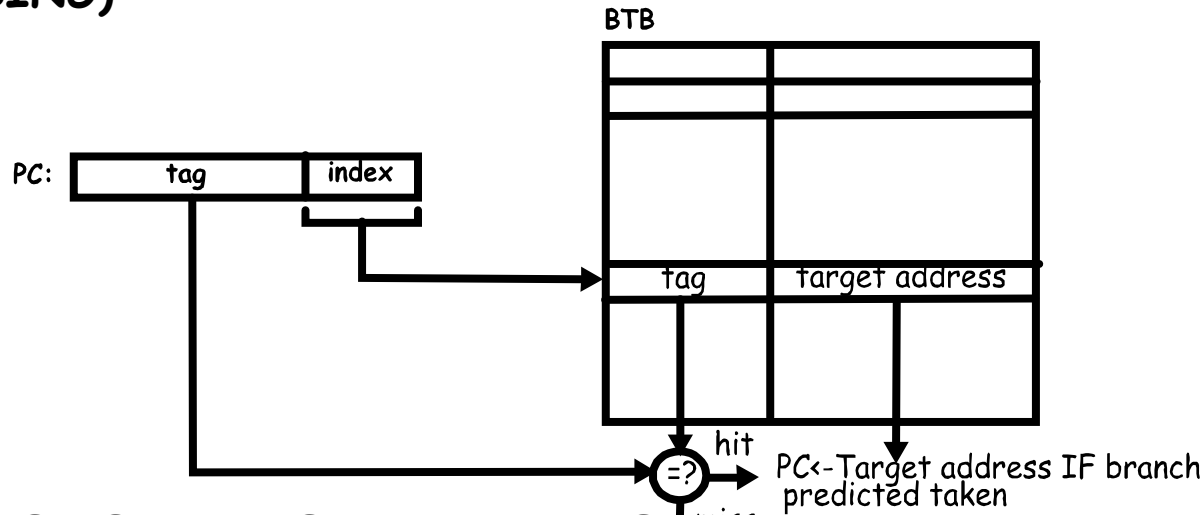
- DIFFERENT BRANCHES CAN BE PREDICTED BETTER WITH DIFFERENT PREDICTORS
- THE BRANCHES IN DIFFERENT PHASES OF A PROGRAM MAY BE PREDICTED BETTER WITH DIFFERENT PREDICTORS
- OR IF TWO PREDICTORS AGREE, THEN THE PROBABILITY THAT THEY ARE RIGHT IS HIGHER THAN IF ANY ONE PREDICTOR IS USED AT A TIME.



- A SELECTOR KEEPS THE TRACK RECORD OF EACH PREDICTOR (GLOBALLY OR FOR EACH BRANCH). THIS CAN BE DONE WITH 1 BIT OR 2 BIT.
- A VOTER TAKES A MAJORITY VOTE OF THE 3 PREDICTORS

BRANCH TARGET BUFFER (BTB)

- TO ELIMINATE THE BRANCH PENALTY
 - NEED TO KNOW THE TARGET ADDRESS BY THE END OF I-FETCH
- THE BTB: CACHE FOR ALL BRANCH TARGET ADDRESSES (NO ALIASING)

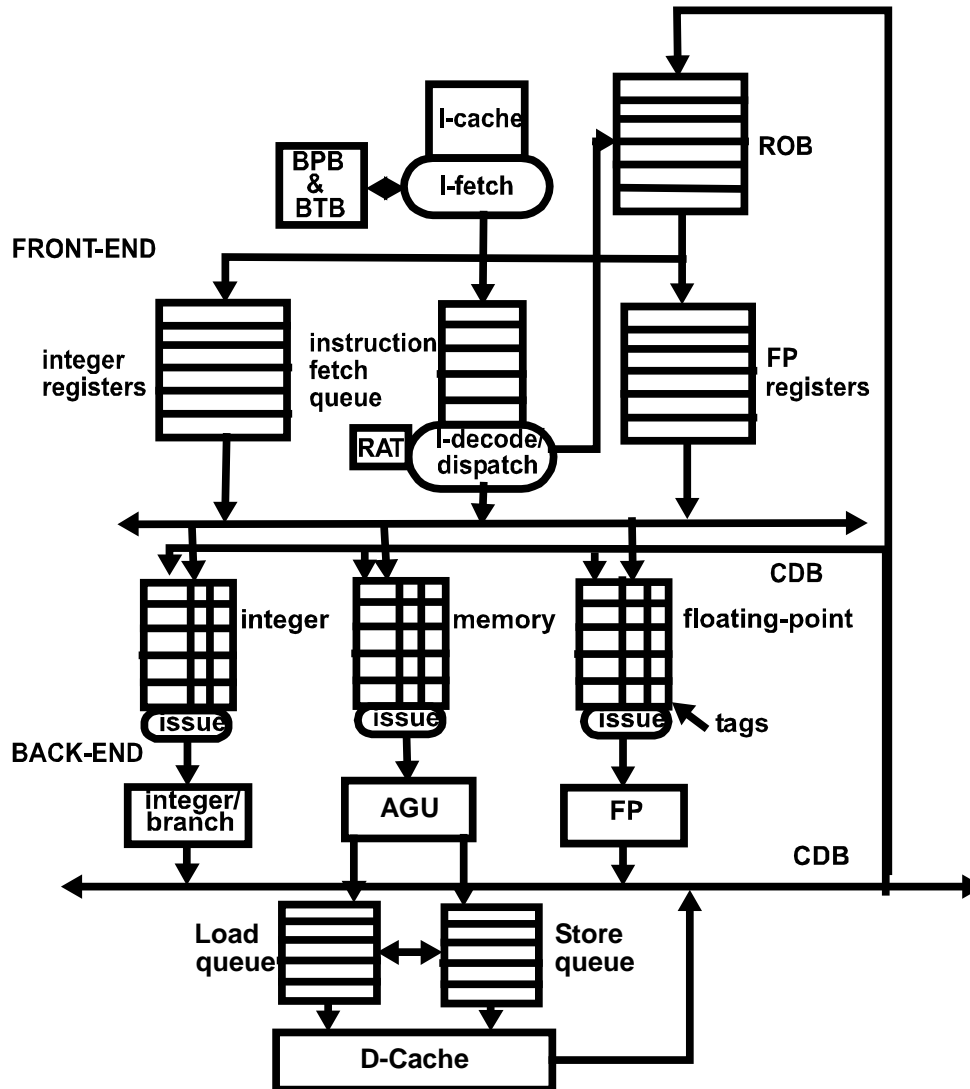


- ACCESSED IN I-FETCH IN PARALLEL WITH INSTRUCTION AND BPB ENTRY
- RELIES ON THE FACT THAT THE TARGET ADDRESS OF A BRANCH NEVER CHANGES
- PREDICTING INDIRECT JUMPS
- PROCEDURE RETURN IS MAJOR CAUSE OF INDIRECT JUMP;
- USE A STACK TO TRACK THE RETURN ADDRESSES OF PROCEDURE CALLS (RAS)

HARDWARE-SUPPORTED SPECULATION

- **COMBINATION OF 3 MAIN IDEAS:**
 - DYNAMIC OoO INSTRUCTION SCHEDULING (TOMASULO)
 - DYNAMIC BRANCH PREDICTION, ALLOWING INSTRUCTION SCHEDULING ACROSS BRANCHES
 - SPECULATIVE EXECUTION: EXECUTE INSTRUCTIONS BEFORE ALL CONTROL DEPENDENCIES ARE RESOLVED.
- **HARDWARE-BASED SPECULATION USES A DATA-FLOW APPROACH: INSTRUCTIONS EXECUTE WHEN THEIR OPERANDS ARE AVAILABLE, ACROSS PREDICTED BRANCHES.**
 - KEY IDEAS:
 - SEPARATE THE COMPLETION OF INSTRUCTION EXECUTION AND THE COMMIT OF ITS RESULT
 - BETWEEN COMPLETION AND COMMIT RESULTS ARE SPECULATIVE
 - COMMIT RESULTS TO REGISTERS AND STORAGE IN PROCESS ORDER
- **WE NEED TO ADD THE FOLLOWING TO TOMASULO:**
 - BRANCH PREDICTION
 - TEMPORARY STORAGE FOR SPECULATIVE RESULTS
 - MECHANISM TO ROLL-BACK EXECUTION WHEN SPECULATION FAILS
- **WE ALSO NEED TO SUPPORT PRECISE EXCEPTIONS**

TOMASULO WITH SPECULATIVE EXECUTION



NEW STRUCTURES:

- REORDER BUFFER (ROB)
- BRANCH PREDICTION BUFFER (BPB)
- BRANCH TARGET BUFFER (BTB)

ROB:

- KEEPS TRACK OF PROCESS ORDER (FIFO)
- HOLDS SPECULATIVE RESULTS
- NO MORE SNOOPING BY REGISTERS

REGISTER VALUES

- PENDING IN BACK-END
- SPECULATIVE IN ROB
- COMMITTED IN THE REGISTER FILE

USE ROB ENTRY # AS TAG TO RENAME REGISTER

STEPS IN SPECULATIVE TOMASULO

- **NORMAL OPERATION (NO MISPREDICTED BRANCH, NO EXCEPTION)**
- **1. I-FETCH**
 - FETCH INSTRUCTION
 - PREDICT BRANCHES AND THEIR TARGET
 - FILL THE INSTRUCTION FETCH Q (IFQ) FOLLOWING THE BRANCH PREDICTION
- **2. I-DECODE/DISPATCH**
 - DECODE OPCODE
 - ALLOCATE 1 ISSUE Q ENTRY + 1 ROB ENTRY + 1 L/S Q ENTRY FOR LOAD/STORE
 - RENAME DESTINATION REGISTER (TAG) WITH ROB ENTRY#, MARK "PENDING" IN RAT
 - SPLIT STORES IN TWO INSTRUCTIONS (ADDRESS + CACHE)
 - FILL INPUT REGISTER OPERAND FIELDS
 - IF VALUE IS MARKED "PENDING" IN RAT FILL OPERAND FIELD WITH TAG (NOT READY)
 - IF MARKED "COMPLETED" IN RAT, FETCH VALUE FROM ROB (READY)
 - IF MARKED "COMMITTED" IN RAT, FETCH VALUE FROM REGISTER FILE (READY)
 - STALL WHILE ROB OR ISSUE Q OR L/S Q (CASE OF L/S) IS FULL
- **3. ISSUE**
 - WAIT IN ISSUE Q UNTIL ALL INPUTS ARE READY (SNOOP THE CDB)
 - ISSUE IN A CYCLE WHEN CONFLICTS FOR FU AND CDB CAN BE AVOIDED

SPECULATIVE TOMASULO

- **4. COMPLETE EXECUTION AND WRITE RESULT**
 - RESULT IS WRITTEN TO THE ROB VIA THE CDB
 - DESTINATION REGISTER IS MARKED "COMPLETED" IN THE RAT
- **5. COMMIT (OR GRADUATE OR RETIRE)**
 - WAIT TO REACH THE HEAD OF THE ROB
 - WRITE RESULT TO REGISTER OR TO MEMORY (STORE)
- **BRANCH MISPREDICTION**
 - ALL INSTRUCTIONS FOLLOWING THE BRANCH IN THE ROB MUST BE FLUSHED
 - WAIT UNTIL THE BRANCH REACHES THE TOP OF THE ROB AND FLUSH THE ROB PLUS FLUSH ALL INSTRUCTIONS IN THE BACK-END
 - INSTRUCTIONS AT THE CORRECT TARGET ARE FETCHED.
- **EXCEPTIONS**
 - EXCEPTIONS ARE FLAGGED IN THE ROB BUT REMAIN "SILENT" UNTIL THE INSTRUCTION IS READY TO RETIRE AT THE TOP OF THE ROB.
 - THE ENTIRE ROB MUST THEN BE FLUSHED
 - SHARE HARDWARE WITH THE MISPREDICTED BRANCH RECOVERY
 - HANDLER INSTRUCTIONS MUST BE FETCHED

SOLVING MEMORY HAZARDS

- **FOR LOAD_s/STORE_s WE USE THE SAME APPROACH AS IN TOMASULO ALGORITHM.**
 - LOAD/STORE INSTRUCTIONS ISSUE TO L/S Q THROUGH THE AGU
 - STORES ARE SPLIT INTO 2 INSTRUCTIONS, ONE COMPUTING THE ADDRESS AND ONE PROPAGATING THE VALUE
 - EACH SUB-INSTRUCTION IS ALLOCATED 1 ISSUE Q ENTRY.
 - ONLY 1 ROB ENTRY ASSOCIATED WITH THE DATA PART OF THE STORE
- **ALL WAW AND WAR HAZARDS ARE AUTOMATICALLY SOLVED**
 - BECAUSE STORES UPDATE THE CACHE IN PROCESS ORDER WHEN THEY REACH THE TOP OF THE REORDER BUFFER.
- **CHECK FOR RAW HAZARDS IN THE LD/ST Q BEFORE SENDING LOAD TO CACHE**
 - LOAD CAN ISSUE TO CACHE AS SOON AS IT REACHES THE L/S Q
 - HOWEVER IF A STORE WITH THE SAME ADDRESS IS IN FRONT OF THE LOAD IN THE LD/ST Q THEN:
 - WAIT UNTIL STORE REACHES THE CACHE (AT THE TOP OF ROB), OR
 - RETURN THE VALUE OF THE STORE WHEN IT IS READY (HOWEVER, THIS MAY AFFECT THE MEMORY CONSISTENCY MODEL)
- **THE BIG PROBLEM IS WHAT TO DO WHEN THERE ARE STORE_s IN FRONT OF THE LOAD WITH UNKNOWN ADDRESS (ADDRESS NOT READY)**

DYNAMIC MEMORY DISAMBIGUATION

- **FIGURING OUT IF TWO ADDRESSES ARE EQUAL TO AVOID HAZARDS**
- **CONSERVATIVE APPROACH**
 - A READY LOAD MUST WAIT IN THE LD/ST Q UNTIL ADDRESSES OF ALL STORES PRECEDING IT IN THE LD/ST Q ARE KNOWN
 - PROBLEM: THE SITUATION WHERE A LOAD DEPENDS ON A STORE IN THE LD/ST Q IS QUITE RARE
- **OPTIMISTIC APPROACH: SPECULATIVE DISAMBIGUATION**
 - USE THE MECHANISMS IN PLACE FOR SPECULATIVE EXECUTION (ROB+ROLLBACK)
 - IF A LOAD IS READY AND STORES WITH UNKNOWN ADDRESSES ARE IN FRONT OF IT IN THE LD/ST Q
 - SPECULATIVELY ASSUME THAT THEIR ADDRESSES ARE DIFFERENT FROM THE LOAD'S ADDRESS
 - ISSUE LOAD TO CACHE
 - LATER, WHEN A STORE'S ADDRESS HAS BEEN COMPUTED AND IS READY, CHECK ALL FOLLOWING LOADS IN THE LD/ST Q
 - IF A LOAD HAS THE SAME ADDRESS AND IS ALREADY COMPLETED, THEN THE LOAD AND ALL FOLLOWING INSTRUCTIONS MUST BE REPLAYED
 - TO DO THIS ROLL BACK THE EXECUTION FROM THE ROB AND I-CACHE, AS IF THE LOAD WAS A MISPREDICTED BRANCH
- **INTERMEDIATE:**
 - KEEP TRACK THAT A LOAD HAS VIOLATED IN THE PAST
 - TREAT THAT LOAD CONSERVATIVELY; TREAT ALL OTHER LOADS OPTIMISTICALLY

EXAMPLE

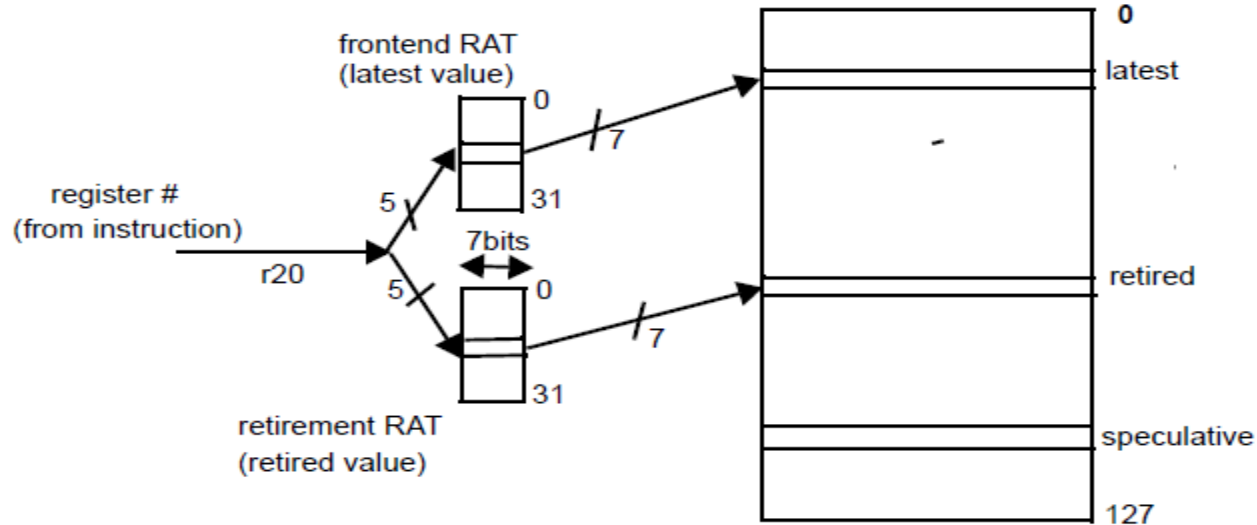
		Dispatch	Issue	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.S F0,0(R1)	1	2	(3)	3	(4)	(5)	6	
I2	L.S F1,0(R2)	2	3	(4)	4	(5)	(6)	7	
I3	ADD.S F2,F1,F0	3	7	(8)	12	--	(13)	14	wait for F1
I4	S.S-A F2,0(R1)	4	5	(6)	6	--	--	--	
I5	S.S-D F2,0(R1)	5	14	(15)	(15)	(16)	--	17	wait for F2
I6	ADDI R1,R1,#4	6	7	(8)	8	--	(9)	18	
I7	ADDI R2,R2,#4	7	8	(9)	9	--	(10)	19	
I8	SUBI R3,R3,#1	8	9	(10)	10	--	(11)	20	
I9	BNEZ R3,Loop	9	12	(13)	13	--	(14)	21	wait for R3
I10	L.S F0,0(R1)	10	12	(13)	13	(14)	(15)	22	CDB conflict with I9
I11	L.S F1,0(R2)	11	13	(14)	14	(15)	(16)	23	issue conflict with I10
I12	ADD.S F2,F1,F0	12	17	(18)	22	--	(23)	24	wait for F1

- **NEW COLUMN: RETIRE**
- **LOAD CAN BE DISPATCHED RIGHT AFTER THE BRANCH**
- **STORE TO CACHE MUST WAIT UNTIL IT REACHES THE TOP OF THE ROB**

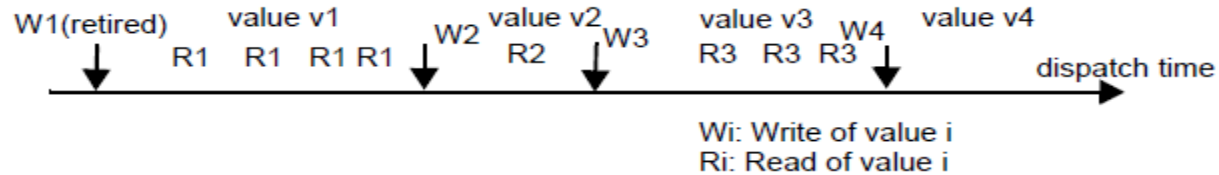
EXPLICIT REGISTER RENAMING

- **ROB ORDERS INSTRUCTION COMMITS AND PROVIDES STORAGE FOR SPECULATIVE REGISTER VALUES UNTIL THEY COMMIT**
- **SPECULATIVE REGISTER VALUES MAY ALSO BE KEPT IN PHYSICAL REGISTERS.**
 - **THEN THE ROLE OF THE ROB IS LIMITED TO ORDERING INSTRUCTION COMMITS**
- **LARGE NUMBER OF PHYSICAL REGISTERS (MORE THAN THE # OF ARCHITECTURAL REGISTERS--THE REGISTERS VISIBLE FROM THE ISA)**
- **ARCHITECTURAL REGISTERS ARE MAPPED TO PHYSICAL REGISTERS DYNAMICALLY**
 - **AT ANY ONE TIME ONE ARCHITECTURAL REGISTER MAY MAP TO MULTIPLE PHYSICAL REGISTERS**
 - **ARCHITECTURAL REGISTERS ARE RENAMED TO PHYSICAL REGISTERS AT DISPATCH**
 - **WHENEVER A NEW VALUE IS STORED IN A REGISTER, A NEW PHYSICAL REGISTER IS ALLOCATED AND THE MAPPING IS CHANGED TO POINT TO THIS LATEST VALUE.**
 - **ONE PHYSICAL REGISTER MUST HOLD THE LATEST COMMITTED (RETIRED) VALUE OF EACH ARCHITECTURAL REGISTER**
 - **PHYSICAL REGISTERS MUST BE RECLAIMED**
 - **ROB ENTRY CARRIES THE MAPPING OF ARCHITECTURAL TO PHYSICAL NUMBER**
- **CAN'T DISPATCH IF ALL PHYSICAL REGISTERS ARE ALLOCATED**

EXPLICIT REGISTER RENAMING



(a)



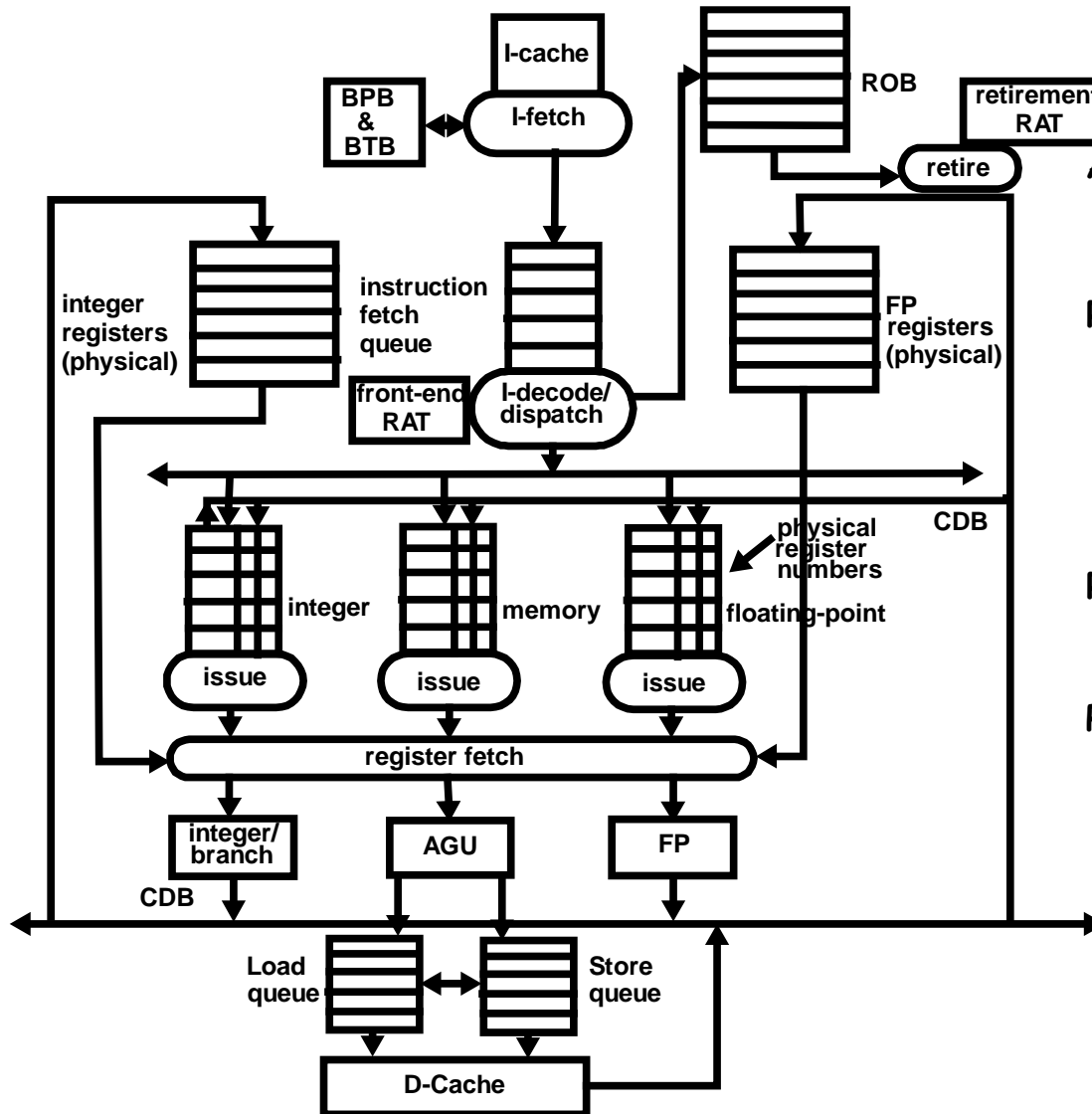
(b)

- FRONTEND RAT POINTS TO MOST RECENT VALUE
- RETIREMENT RAT POINTS TO MOST RECENT *RETIRED* VALUE
 - COULD BE THE SAME AS MOST RECENT VALUE
- MORE SPECULATIVE VALUES OF THE SAME ARCHITECTURAL REGISTER MAY BE IN THE REGISTER FILE
- "UPDATING REGISTER" NOW MEANS "UPDATING THE POINTER IN THE RETIREMENT RAT"

EXPLICIT REGISTER RENAMING

- **WHEN AN INSTRUCTION IS DISPATCHED ITS OPERANDS ARE RENAMED**
 - A PHYSICAL REGISTER IS ALLOCATED TO THE DESTINATION REGISTER
 - THE FRONTEND RAT IS UPDATED
 - THE PHYSICAL REGISTER NUMBER MAPPED BY THE FRONTEND IS NOW THE TAG USED IN TOMASULO ALGORITHM (NOT THE ROB ENTRY #)
 - INPUT REGISTER OPERANDS ARE MAPPED TO THEIR PHYSICAL REGISTER THROUGH THE FRONTEND RAT
 - IF VALUE IS READY IT IS DISPATCHED TO THE ISSUE Q. OTHERWISE THE PHYSICAL REGISTER NUMBER IS DISPATCHED (NOT_READY)
 - WHEN AN INSTRUCTION RETIRES ITS DESTINATION PHYSICAL REGISTER HAS THE RETIRED VALUE AND THE RETIREMENT RAT IS UPDATED
 - PREVIOUS PHYSICAL REGISTER MAPPED BY THE RETIREMENT RAT IS RECLAIMED
- **WHEN A BRANCH IS MISPREDICTED, WE NEED TO FLUSH THE FOLLOWING INSTRUCTIONS BUT WE MUST ALSO RESTORE THE FRONTEND RAT TO ITS CONTENT WHEN THE BRANCH WAS DISPATCHED.**
 - SAVING THE MAP ON EACH BRANCH DISPATCH AND RESTORING IT, OR
 - REBUILDING THE MAP ONE BY ONE FROM THE RETIREMENT RAT BY TRAVERSING THE ROB BACKWARDS FROM TOP OF ROB TO THE BRANCH OR
 - DEMAPPING REGISTERS ONE BY ONE BY TRAVERSING THE ROB FORWARDS FROM BOTTOM TO BRANCH STARTING WITH THE CURRENT FRONTEND (NEEDS OLD MAPPINGS)
 - RETIREMENT RAT IS NOT A PROBLEM SINCE IT MAPS COMMITTED VALUES
- **SAME FOR EXCEPTIONS**

REGISTER FETCH AFTER ISSUE



AT DISPATCH INPUT REGISTER # (NOT ITS VALUE) IS DISPATCHED WITH READY BIT

ROLE OF THE CDB:

- TRANSFER VALUES TO REGISTER
- AWAKE INSTRUCTIONS BY SETTING READY BITS IN ISSUE Qs

REGISTERS ARE FETCHED RIGHT BEFORE EXECUTION

PROBLEMS:

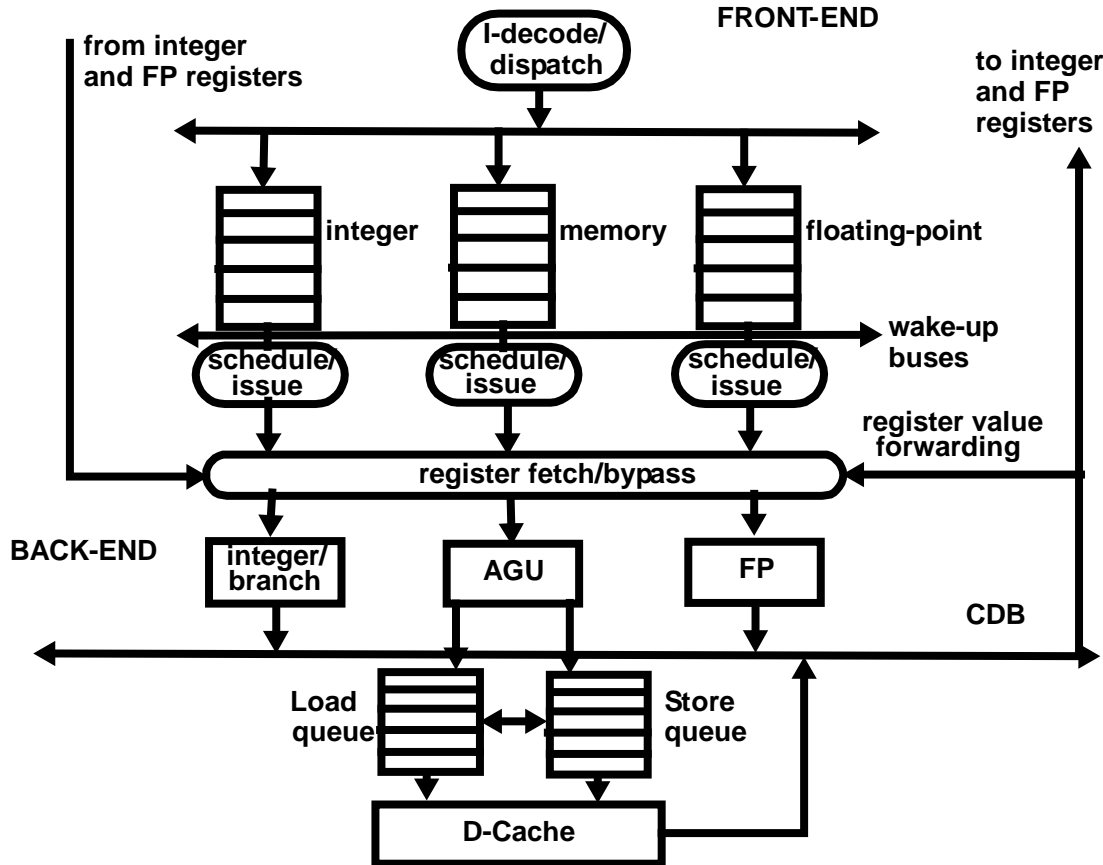
- EFFECTIVE LATENCY MUCH HIGHER
- SIMILAR PROBLEM IN TOMASULO

EXAMPLE

		Dispatch	Issue	Register fetch	Exec start	Exec complete	Cache	CDB	Retire	COMMENTS
I1	L.S F0,0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L.S F1,0(R2)	2	3	4	(5)	5	(6)	(7)	8	
I3	ADD.S F2,F1,F0	3	8	9	(10)	14	--	(15)	16	wait for F1
I4	S.S-A F2,0(R1)	4	5	6	(7)	7	--	--		
I5	S.S-D F2,0(R1)	5	16	17	(18)	(18)	(19)	--	20	wait for F2
I6	ADDI R1,R1,#4	6	7	8	(9)	9	--	(10)	21	
I7	ADDI R2,R2,#4	7	8	9	(10)	10	--	(11)	22	
I8	SUBI R3,R3,#1	8	9	10	(11)	11	--	(12)	23	
I9	BNEZ R3,Loop	9	13	14	(15)	15	--	(16)	24	wait for R3
I10	L.S F0,0(R1)	10	13	14	(15)	15	(16)	(17)	25	CDB conflicts with I3&I9
I11	L.S F1,0(R2)	11	14	15	(16)	16	(17)	(18)	26	CDB conflict with I10
I12	ADD.S F2,F1,F0	12	18	19	(20)	24	--	(25)	27	wait for F1

- WE HAVE ASSUMED THAT REGISTER FILES HAVE ENOUGH PORTS TO AVOID THE NEED TO RESERVE THEM.

SPECULATIVE INSTRUCTION SCHEDULING

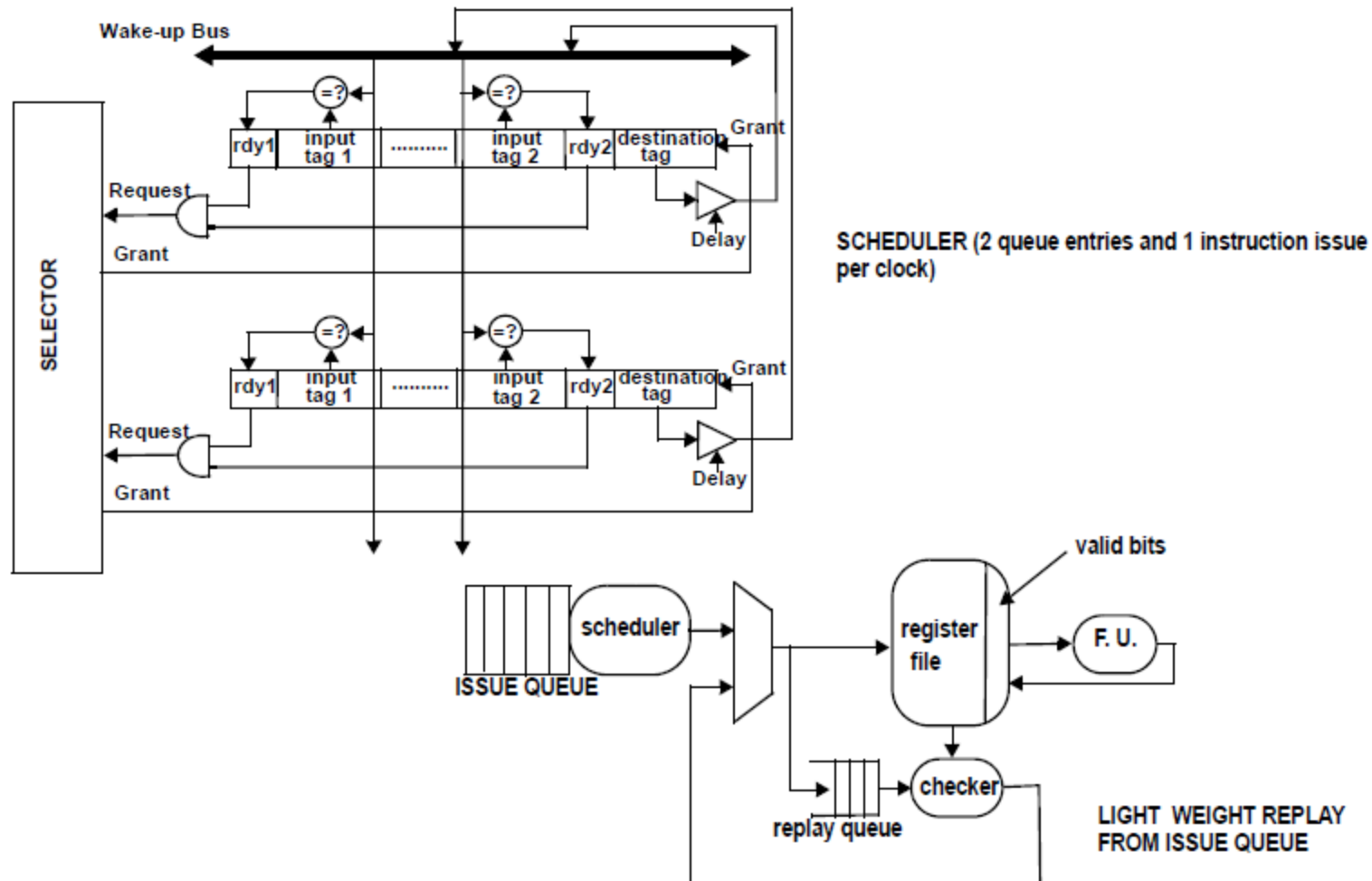


THE SCHEDULER WAKES UP OPERANDS IN ISSUE Qs

- IT SETS THE READY BITS OF WAITING OPS AFTER A NUMBER OF CYCLES EQUAL TO THE LATENCY OF OPERATION
 - DOES THAT EVERY TIME IT ISSUES AN INSTRUCTION
 - DEPENDENT INSTRUCTIONS ARE AWAKENED AND CAN BE SCHEDULED AFTER ALL THEIR READY BITS ARE SET
 - VALUES ARE FORWARDED
- ## SOME INSTRUCTIONS HAVE VARIABLE LATENCIES (E.G., CACHE ACCESSES)
- HIT/MISS

- CDB IS NOW A SIMPLE BUS: FORWARDS AND STORES IN REGISTERS
- WHEN SPECULATION FAILS THE SCHEDULE MUST BE REPAIRED
 - TREAT THE SCHEDULE VIOLATION AS A MISPREDICTED BRANCH
 - OR USE A LIGHT WEIGHT REPLAY MECHANISM

SPECULATIVE INSTRUCTION SCHEDULING



EXAMPLE

Issue ADD.S
EVEN BEFORE
GETTING F1

		Dispatch	Issue	Register fetch	Exec start	Exec complete	Cache	CDB	Retire	Comments
I1	L.S F0,0(R1)	1	2	3	(4)	4	(5)	(6)	7	
I2	L.S F1,0(R2)	2	3	4	(5)	5	(6)	(7)	8	
I3	ADD.S F2,F1,F0	3	5	6	(7)	11	--	(12)	13	wait for F1
I4	S.S-A F2,0(R1)	4	5	6	(7)	7	--	--	--	
I5	S.S-D F2,0(R1)	5	10	11	(12)	12	(13)	--	14	wait for F2
I6	ADDI R1,R1,#4	6	7	8	(9)	9	--	(10)	15	
I7	ADDI R2,R2,#4	7	8	9	(10)	10	--	(11)	16	
I8	SUBI R3,R3,#1	8	10	11	(12)	12	--	(13)	17	CDB conflict with I3
I9	BNEZ R3,Loop	9	11	12	(13)	13	--	(14)	18	Issue conflict with I8
I10	L.S F0,0(R1)	10	11	12	(13)	13	(14)	(15)	19	
I11	L.S F1,0(R2)	11	12	13	(14)	14	(15)	(16)	20	CDB conflict with I9
I12	ADD.S F2,F1,F0	12	14	15	(16)	20	--	(21)	22	wait for F1

VALUE PREDICTION

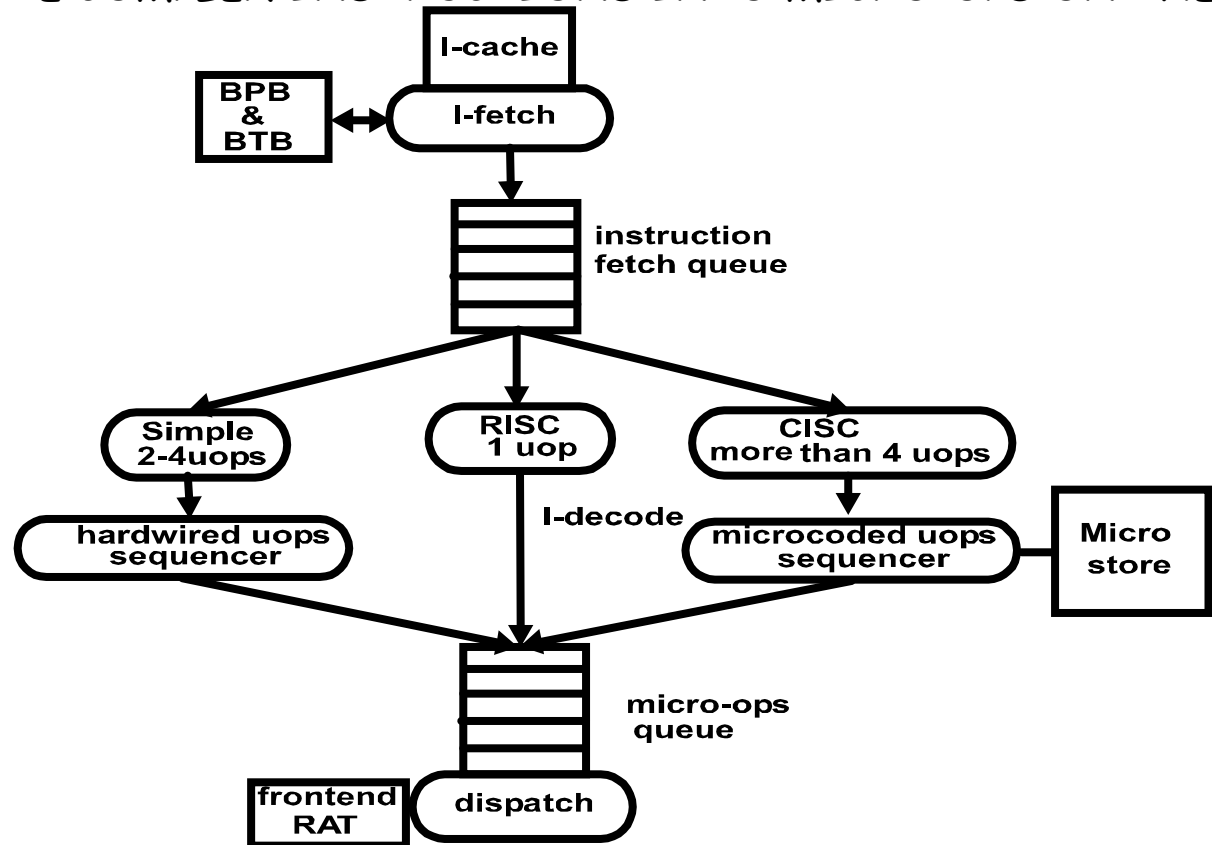
- **SO FAR A DEPENDENT INSTRUCTION MUST WAIT UNTIL ITS VALUES ARE READY TO START EXECUTING**
- **HOWEVER MANY COMPUTED VALUES ARE PREDICTABLE:**
 - MANY INSTRUCTIONS ALWAYS PRODUCE THE SAME VALUE
 - OR ONE VALUE AMONG A SMALL SET OF RECENT VALUES
 - OR VALUES THAT INCREASE OR DECREASE WITH A STRIDE
- **START THE EXECUTION OF AN INSTRUCTION SPECULATIVELY BY PREDICTING THE VALUES OF ITS INPUT OPERANDS EVEN IF THEY ARE NOT READY.**
 - GREAT FOR INSTRUCTIONS DEPENDING ON LONG LATENCY LOADS
- **WHEN A LONG LATENCY INSTRUCTION IS DETECTED IN THE DECODE STAGE ITS VALUE IS PREDICTED AND STORED IN THE DESTINATION PHYSICAL REGISTER AT DISPATCH**
 - USE PREDICTION TABLES ACCESSED BY PC
 - THE PREDICTED VALUE CAN BE USED SPECULATIVELY BY DEPENDENT INSTRUCTIONS READING THE REGISTER
 - DEPENDENT INSTRUCTIONS CAN ISSUE BEFORE THEIR PARENT
 - WHEN THE PARENT COMPLETES, VALUE IS COMPARED WITH THE PREDICTED VALUE
 - IF EQUAL, THE EXECUTION CONTINUES UNDISTURBED
 - IF NOT EQUAL, THE COMPUTED VALUE IS STORED IN REGISTER AND
 - INSTRUCTIONS FOLLOWING THE MISPREDICTED VALUE ARE FLUSHED THROUGH THE ROB AND I-CACHE
- **AVOID PREDICTION FOR INSTRUCTIONS THAT VIOLATE OFTEN**

DISPATCHING MULTIPLE INSTRUCTIONS PER CLOCK

- **GOAL: EXPLOITING MORE ILP SO THAT $CPI < 1$ OR $IPC > 1$.**
- **ASSUME A 16-WAY OoO PROCESSOR**
- **PROBLEM WITH I-FETCH:**
 - MUST FILL THE IFQ AT THE SAME RATE
 - 16 INSTRUCTIONS CONTAIN 2-3 BRANCHES
 - BRANCHES MUST BE PREDICTED AND CODE MUST BE FETCHED PIECEMEAL FROM I-CACHE ALL IN 1 CLOCK
 - USE TRACE CACHE
- **PROBLEM WITH DISPATCH**
 - MUST RENAME MANY INSTRUCTIONS IN 1 CYCLE
 - THIS IS A SEQUENTIAL PROCESS. MUST BE DONE IN THREAD ORDER
 - FRONTEND RAT MUST BE UPDATED CONSISTENTLY WITH DISPATCHING EACH INSTRUCTION ONE BY ONE
 - FRONTEND RAT MUST HAVE ENOUGH BANDWIDTH
 - UP TO 16 PHYSICAL REGISTERS MUST BE ALLOCATED IN ONE CLOCK
 - UP TO 32 ARCHITECTURAL INPUT REGISTERS MUST BE MAPPED
- **PROBLEM WITH BACKEND**
 - NO BOTTLENECK
 - ENOUGH FUs
 - ENOUGH CDB BANDWIDTH
 - ENOUGH REGISTER FILE BANDWIDTH TO STORE VALUES
 - ENOUGH RETIREMENT BANDWIDTH

DEALING WITH COMPLEX ISAs

- SO FAR RISC ISA
- FOR COMPLEX ISAs: USE MICROCODE
 - IDENTIFY A CORE SET OF RISC INSTRUCTIONS
 - MAKE THEM THE MICRO-CODE; DON'T SLOW THEM DOWN (COMMON CASE)
 - TRANSLATE COMPLEX INSTRUCTIONS INTO MICRO-OPS ON THE FLY



INTEL PENTIUM III

- **Several processor implementations based on the same processor architecture**

Processor	ship	clock	L1 cache	L2 cache
Pentium Pro	1995	100-200MHz	8K/8K	256-1024KB
Pentium II	1998	233-450MHz	16K/16K	256-512KB
Pentium II Xeon	1999	400-450MHz	16K/16K	512-2048KB
Celeron	1999	500-900MHz	16K/16K	128KB
Pentium III	1999	450-1100MHz	16K/16K	256-512KB
Pentium III Xeon	2000	700-900MHz	16K/16K	1024-2048KB

- **Dynamically scheduled**
- **RISC (load/store) core**
- **Complex instructions are translated on the fly into instructions of the RISC core.**
- **If an instruction needs more than four microops then it is implemented by a microcoded sequence of microops.**
- **Maximum number of microops is 6 per clock**
- **Microops (RISC-like instructions) are issued and executed OOO with speculation**
- **14 stages**

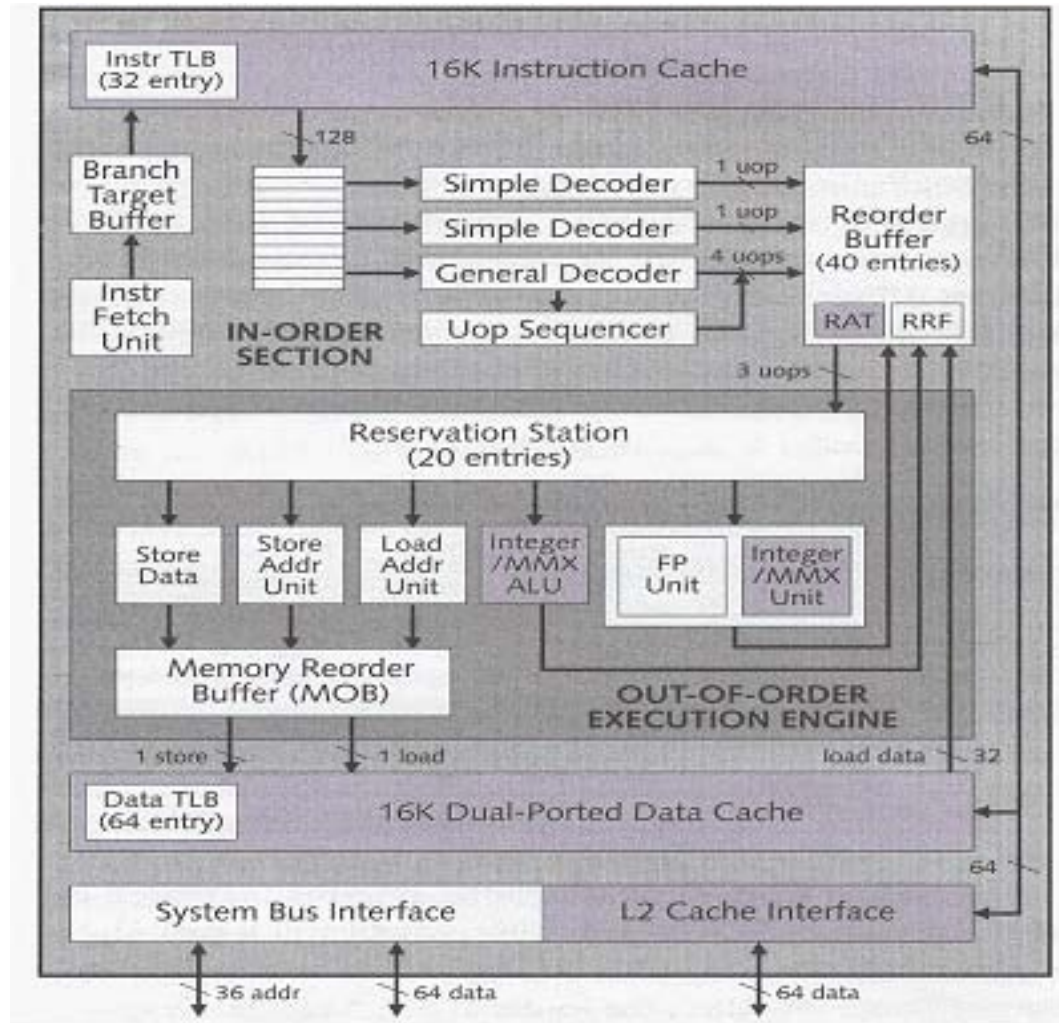
INTEL PENTIUM III

- 8 stages for fetch, decode and dispatch. Includes 512-entry 2-level branch predictor.
- Also, 40 virtual registers and 20 reservation stations (shared)
- 3 stages for execution

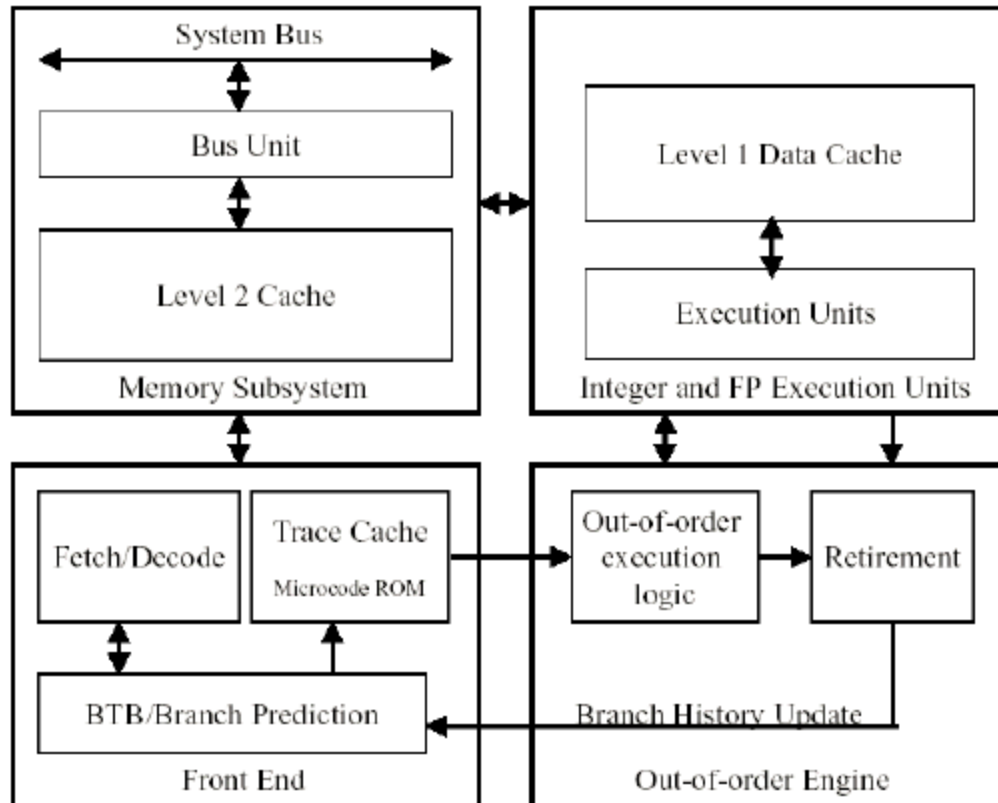
Instruction	latency	repeat
Int. ALU	0	1
Load	2	1
Int Mult	3	1
FP add	2	1
FP Mult	4	2
FP Div	31	32

- Latencies and repeat intervals

INTEL PENTIUM III



INTEL PENTIUM 4



INTEL PENTIUM 4

- 42 million transistors
- 217mm²
- 55 watts at 1.5GHz
- 4 Parts:
- Front end:
 - Trace cache + instruction cache
 - 32-bit instructions are decoded into up to 4 microops
 - pointer in ROM if more than 4 microops
 - microops are buffered in a microop queue (in order)
 - Trace cache predictor (512entries) + BTB of 4K entries
- OoO logic
 - allocates physical registers (128 int and 128 fp) and ROB entries (126-up to 48 loads and 24 stores) for 3microops in each cycle(stalls if structural hazard).
 - Register renaming supported by RAT
 - microops are then queued in FIFO order in two queues: memory queue/non-memory queue (same as reservation stations)
 - when operands are available, microops are dispatched up to 6 at a time
 - Register operands are fetched between the scheduler and the execution unit (same as scoreboard)
 - speculative dispatching of dependent instructions to cut the latency

INTEL PENTIUM 4

- Execution Unit
- 1 LD, 1 ST, 3 INT and 2 FP/MMX
- Memory system
- L1 8KB 4-way 128B
- No I-cache
- L2 Unified 256KB 8-way 128B

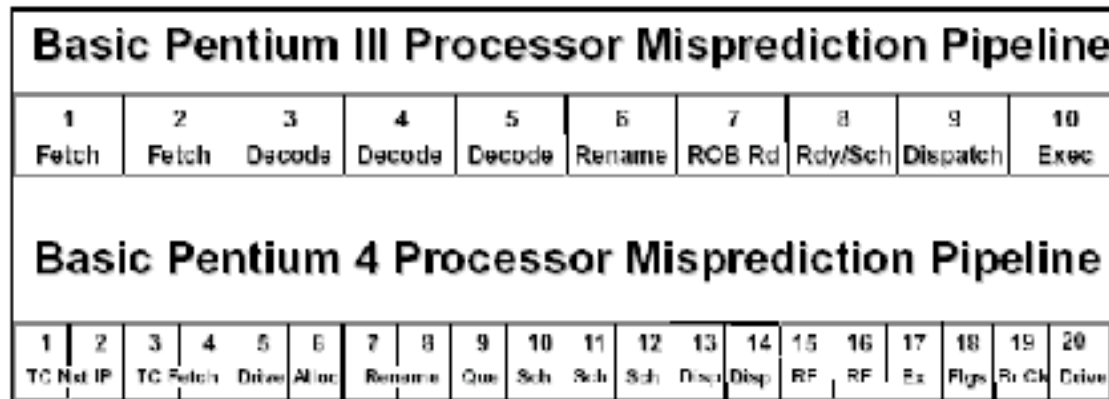


Figure 3: Misprediction Pipeline

INTEL PENTIUM 4

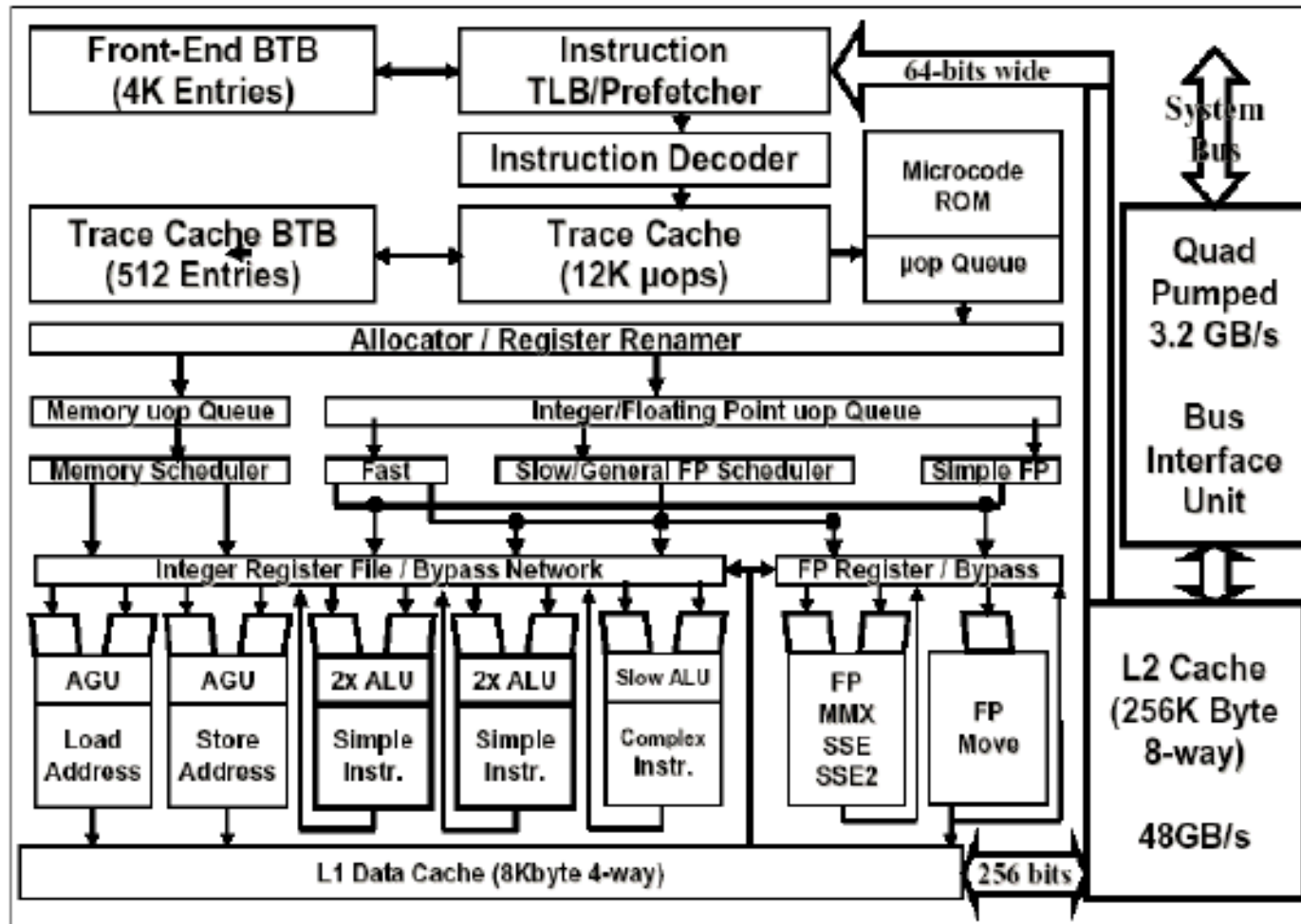


Figure 4: Pentium[®] 4 processor microarchitecture

PENTIUM III vs PENTIUM 4 REGISTER RENAMING

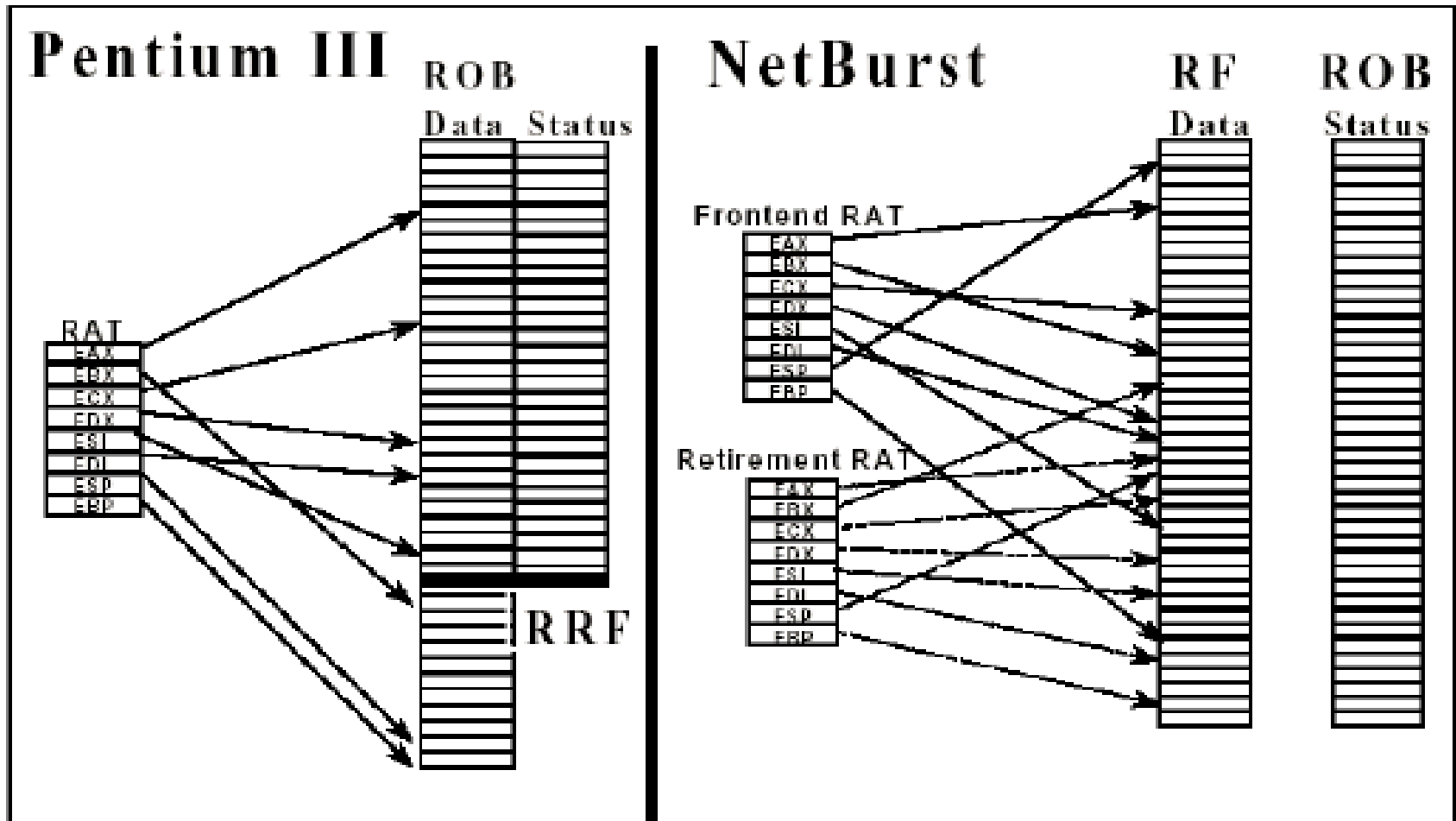


Figure 5: Pentium® III vs. Pentium® 4 processor register allocation

VLIW-EPIC-VECTOR

- LIW/VLIW
- VLIW SCHEDULING
- HARDWARE SUPPORT FOR MORE ILP
- EPIC ARCHITECTURE: INTEL IA-64
- VECTOR PROCESSORS

DUALITY OF DYNAMIC AND STATIC TECHNIQUES

- Instruction scheduling: Compiler moves instructions.
Same issues: data-flow and exception model.
- Software register renaming for WAW and WAR hazards.
- RAW hazards can be solved by hardware or by software
- Memory disambiguation must be done by the compiler
- Branch prediction scheme: static prediction
- Speculation: speculate based on static branch prediction. Test dynamically and jump to patch-up/recovery code if the speculation failed.
- Sometimes there is no need to speculate because the compiler knows the structure of the program (e.g. loops)

PROGRAM EXAMPLE

INSTR. PRODUCING RESULT	INSTR. USING RESULT	LATENCY
FP ALU op	FP ALU op	2
LOAD DOUBLE	FP ALU op	1
STORE DOUBLE	LOAD DOUBLE	0
INT LOAD	INT ALU op/Branch	1
BRANCH DELAY SLOT	N/A	2

CONSIDER THE FOLLOWING PROGRAM:

FOR (i = 1000; i > 0; i = i-1)

x[i] = x[i] + s

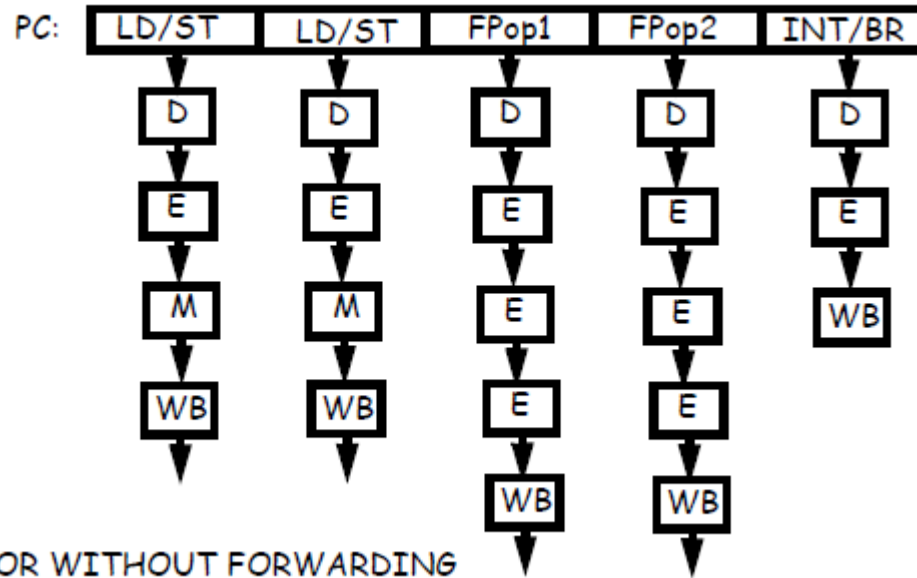
which is compiled into:

```
Loop:  L.D      F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        SUBI    R1, R1, #8
        BNE     R1, R2, Loop
        NOOP
        NOOP
```

BASIC LIW/VLIW

- **MULTIPLE *INDEPENDENT* RISC INSTRUCTIONS --called ops--**
 - ARE PACKAGED IN ONE LIW or VLIW--CALLED INSTRUCTION
- **INDEPENDENT FUNCTIONAL UNITS WITH NO HAZARD DETECTION**
- **MAY HAVE SOME FORWARDING TO REDUCE LATENCIES OF OPERATION**
- **COMPILER IS RESPONSIBLE FOR DISPATCHING/SCHEDULING INSTRUCTIONS**
- **EXAMPLE:**
 - 1 INT OR BRANCH op
 - 2 FP ops
 - 2 MEMORY REFERENCES ops
 - EACH op TAKES 16 TO 24 BITS
 - TOTAL INSTRUCTION IS 112-168 BITS
- **USE LOCAL AND GLOBAL COMPILER SCHEDULING ALGORITHMS**

VLIW ARCHITECTURE



- **Cyclic scheduling:**
Loop unrolling or software pipelining.
Applicable to loops
- **Non-cyclic scheduling**
For straight-line code with branches

VLIW--LOOP UNROLLING

UNROLL LOOP 7 TIMES--VLIW PROGRAM:

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	L.D F6,-8(R1)			
2	L.D F10,-16(R1)	L.D F14,-24(R1)			
3	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
4	L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
5			ADD.D F20,F18,F2	ADD.D F24,F22,F2	
6	S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2		SUBI R1,R1,#56
7	S.D 40(R1),F12	S.D 32(R1),F16			BNE R1,R2,Clock1
8	S.D 24(R1),F20	S.D 16(R1),F24			
9	S.D 8(R1),F28				

- **THE BRANCH IS DELAYED BY TWO INSTRUCTIONS TO AVOID FLUSHING**
- **PROBLEMS**
 - CODE SIZE
 - EMPTY SLOTS
 - REGISTER PRESSURE
 - LOCKSTEP
 - BINARY COMPATIBILITY (BINARY TRANSLATION/EMULATION)
 - COST OF BRANCHES
 - ILP ILP ILP ILP...

VLIW--SOFTWARE PIPELINING

Loop: L.D F0,0(R1) O1
 ADD.D F4,F0,F2 O2
 S.D 0(R1),F4 O3

1 LOOP ITERATION PER CLOCK

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

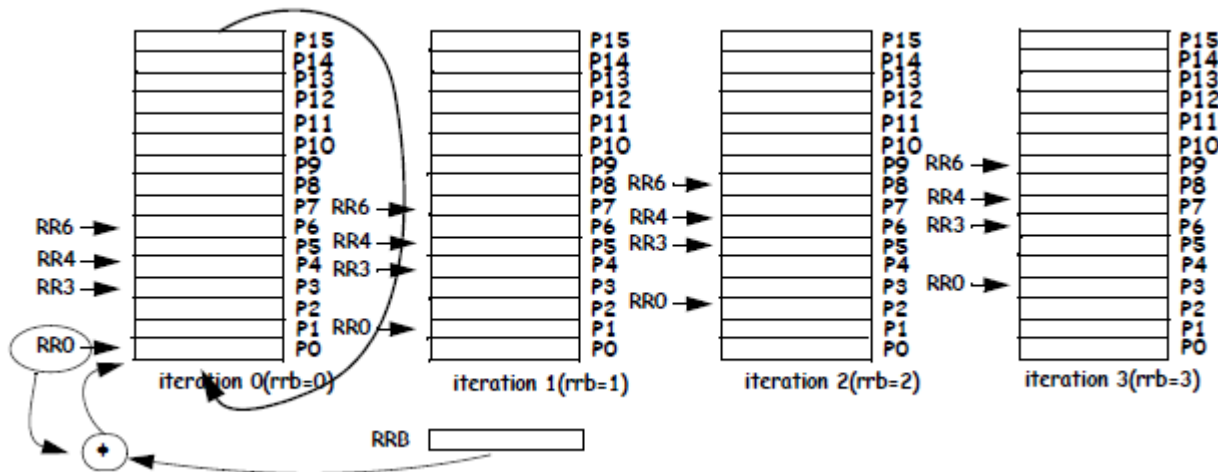
Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	S.D 40(R1),F4	ADD.D F4,F0,F2	NOOP	

RAW HAZARDS ARE SOLVED. WAW HAZARDS REMAIN.

TO SOLVE THEM, USE ROTATING REGISTERS.

REGISTER ROTATION IS A FORM OF REGISTER RENAMING

VLIW--ROTATING REGISTERS



- RR_i is Rotating register i . RR_i maps to Physical register $(RRB+i) \bmod 16$. RRB is incremented by 1 at every iteration
- WITH ROTATING REGISTERS WE REWRITE THE KERNEL CODE:
- RR_4 maps to P_6 (currently RR_6) 2 cycles (iterations) later and RR_0 maps to P_3 (currently RR_3) 3 cycles (iterations) later

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D $RR_6, 0(R_1)$	S.D $40(R_1), RR_0$	ADD.D RR_3, RR_4, F_2	NOOP	

- The loop instruction is not included (to simplify), but it would increment RRB every time it is executed

VLIW--SLOT CONFLICTS

- Now assume that we have 3 slots: 1 LD/ST, 1 FP OP and 1BR/INT
- We have two memory accesses per iteration. Thus maximum rate is one iteration every two cycles.

	ITE1	ITE2	ITE3	ITE4
INST1	O1			
INST2	--			
INST3	O2	O1		
INST4	--	--		
INST5	--	O2	O1	
INST6	O3	--	--	
INST7		--	O2	O1
INST8		O3	--	--

- **KERNEL CODE: STORE IS TWO ITERATIONS BEHIND THE LOAD; SUB MOVED UP**

Clock	MemOp1	FPOp1	Int/BROp
1	L.D F0,0(R1)	ADD.D F4,F0,F2	
2	S.D 24(R1),F4	NOOP	

- **WITH ROTATING REGISTERS**

Clock	MemOp1	FPOp1	Int/BROp
1	L.D RR3,0(R1)	ADD.D RR1,RR2,F2	
2	S.D 24(R1),RR0	NOOP	

VLIW--LOOP-CARRIED DEPENDENCIES

for (i = 0; i < N; i++)

{A[i+2] = A[i] +1;

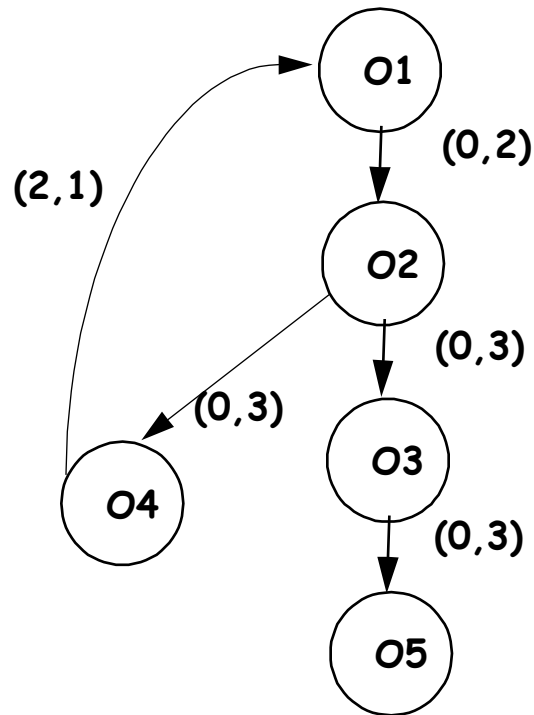
B[i] = A[i+2]+1;}

- **In this statement, A[i] and B[i] are vectors of double-precision FP numbers (64 bits).**
- **The MIPS code is:**

Loop:	L.D F0, 0(R2)	O1
	ADD.D F3, F0, F1	O2
	ADD.D F4, F3, F1	O3
	S.D F3, 16(R2)	O4
	S.D F4, 0(R3)	O5
	ADDI R2, R2, 8	
	ADDI R3, R3, 8	
	BNE R2, R4, Loop	

- **THERE IS A LOOP CARRIED DEPENDENCY ON MEMORY**
 - **Cannot be solved with rotating registers.**
- **LET's LOOK AT THE DATA DEPENDENCY GRAPH.**

VLIW--LOOP-CARRIED DEPENDENCIES



Each node is an op in the loop body

Each arc connects two dependent ops

- for registers and memory locations

Each arc is labelled with (diff, min)

- diff is the number of iterations between the 2 ops

- min is the minimum number of clocks to avoid hazards due to data dependencies

We assume that a load can return the value of a store with the same address in the next clock

- The cycle means that O1 (the load) cannot be scheduled earlier than right after the store (O4) scheduled two iterations before.
OR
- The load cannot be scheduled earlier than 6 cycles before the load scheduled two iterations before
- This further implies that the maximum rate at which iterations can be scheduled is 2 per 6 clocks or 1 per 3 clocks

VLIW--LOOP CARRIED DEPENDENCIES

	ITE1	ITE2	ITE3	ITE4	ITE5
INST1	O1		PROLOGUE		
INST2	--				
INST3	O2				
INST4	--	O1		KERNEL	
INST5	--	--			
INST6	O3, O4	O2			
INST7	--	--	O1		
INST8	--	--	--		
INST9	O5	O3, O4	O2	EPILOGUE	
INST10	--		--	O1	
INST11		--	--	--	
INST12		O5	O3, O4	O2	
INST13		--		--	O1
INST14					
INST15			O5	O3, O4	O2

- **KERNEL (USING 2 LD/SD, 2 FP AND 1 BR/INT SLOTS)**

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	NOOP	L.D F0,0(R2)	NOOP	NOOP	BNE R2,R4,CLK1
2	NOOP	NOOP	NOOP	NOOP	ADDI R2,R2,#8
3	S.D F3,16(R2)	S.D F4,0(R3)	ADD.D F3,F0,F1	ADD.D F4,F3,F1	ADDI R3,R3,#8

VLIW--LOOP-CARRIED DEPENDENCIES

- S.D F3, 16(R2) (O4) is behind L.D F0 (O1) by 1 iteration; ADDI R2 is moved up; so we need to subtract 16 from its displacement
- S.D F4,0(R3) (O5) is behind L.D F0 (O1) by 2 iterations; so we need subtract 16 from its displacement.

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	NOOP	L.D F0,0(R2)	NOOP	NOOP	BNE R2,R4,CLK1
2	NOOP	NOOP	NOOP	NOOP	ADDI R2,R2,#8
3	S.D F3,0(R2)	S.D F4,-16(R3)	ADD.D F3,F0,F1	ADD.D F4,F3,F1	ADDI R3,R3,#8

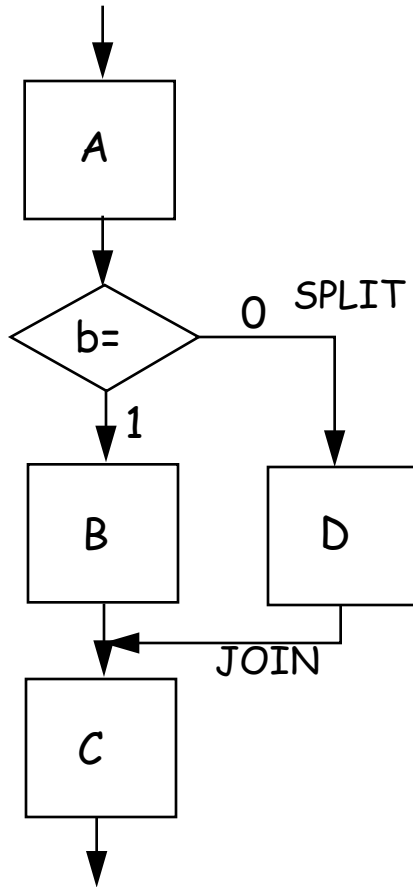
- WITH ROTATING REGISTERS, THE VLIW PROGRAM BECOMES:

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	NOOP	L.D F4,0(R2)	NOOP	NOOP	BNE R2,R4,CLK1
2	NOOP	NOOP	NOOP	NOOP	ADDI R2,R2,#8
3	S.D RR2,0(R2)	S.D RR0,-16(R3)	ADD.D RR3,F4,F1	ADD.D RR1,RR2,F1	ADDI R3,R3,#8

VLIW--SUMMARY OF SOFTWARE PIPELINING

- 1) Build the DDG of the loop
- 2) Compute the minimum Initiation Interval (II) by considering both cycles in the DDG and slot conflicts
 - For every cycle in the DDG
 - compute the number of clocks in the cycle (MIN)
 - compute the number of iterations in the cycle (DIFF)
 - compute MIN/DIFF
 - Find the MAXIMUM of all MIN/DIFF over all cycles in the DDG (II1)
 - For each type of Ops
 - divide the number of Ops in one iteration of the loop by the number of slots for that type of Op
 - Find the MAXIMUM of the resulting number over all types of slots (II2) and round it up
 - Take the MAX of II1 and II2. Call it MINII.
- 3) Layout the schedule of consecutive iterations of the loop separated by MINII cycles. Find one repetitive kernel.
- 4) Write VLIW code for kernel
- 5) If no repetitive kernel or no program kernel can be found add 1 to MINII and repeat 3
- 5) Adjust displacements in address fields
- 6) Assign Rotating Registers to variable register operands

NON-CYCLIC SCHEDULING



A,B,C,AND D ARE BASIC BLOCK
PREDICT THE BRANCH STATICALLY
LET A,B,C BE THE MOST LIKELY TRACE
WE CAN SCHEDULE THE CODE IN THAT
TRACE AS IF IT WAS A BASIC BLOCK

- SOME RESTRICTIONS APPLY (stores)

IF IT TURNS OUT THAT THE BRANCH
WAS TAKEN SO THAT D WAS SUPPOSED
TO BE EXECUTED INSTEAD OF B

- WE NEED TO REPAIR THE TRACE
- WE NEED TO EXECUTE D
- THIS IS DONE BY EXECUTING ANOTHER
TRACE CONTAINING D PLUS SOME
COMPENSATION CODE.

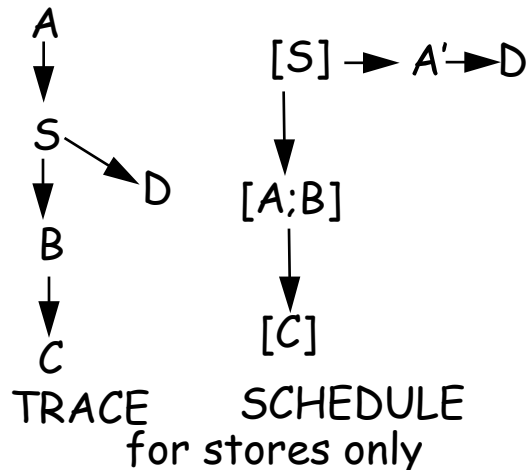
CALLED "TRACE SCHEDULING"

**WORKS IF BRANCH IS PREDICTED RIGHT
MOST OF THE TIME**

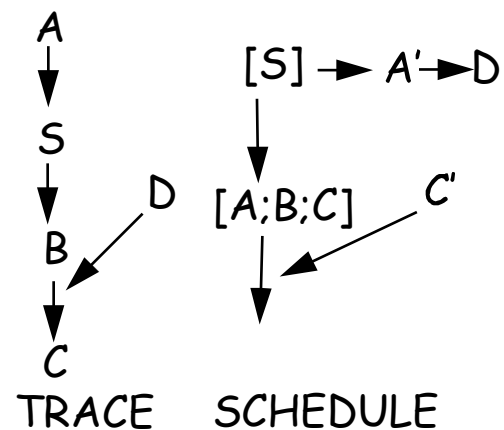
TRACE SCHEDULING

- **COMPENSATION CODE FOR NON-SPECULATIVE CODE MOTION:**

DOWN ACROSS SPLIT



UP ACROSS JOIN



- **SPECULATIVE CODE MOTION**

- **MOVING CODE UP ACROSS SPLIT**
 - BIG NO-NO FOR STORES
 - SHOULD NOT AFFECT THE OFF-TRACE PATH
 - OTHERWISE MUST UNDO THE RESULT OF THE MOVED INSTRUCTION ON THAT PATH
 - MAJOR SOURCE OF CODE MOTION AND SPEEDUP
- WE DON'T MOVE CODE DOWN ACROSS A JOIN AND WE DON'T MOVE SPLIT ACROSS OTHER SPLIT (SPLITS REMAIN IN PROGRAM ORDER)

TRACE SCHEDULING EXAMPLE

	LD R4,0(R1)	A
	LD R5,0(R2)	B
	BEQ R5,R4,LAB1	S
	ADDI R6,R4,5	G
	SD R6,0(R3)	H
	JMP LAB2	I
LAB1:	ADDI R7,R5,2	C
	SD R7,0(R3)	D
LAB2:	ADD R8,R6,R4	E
	SD R8,0(R1)	F

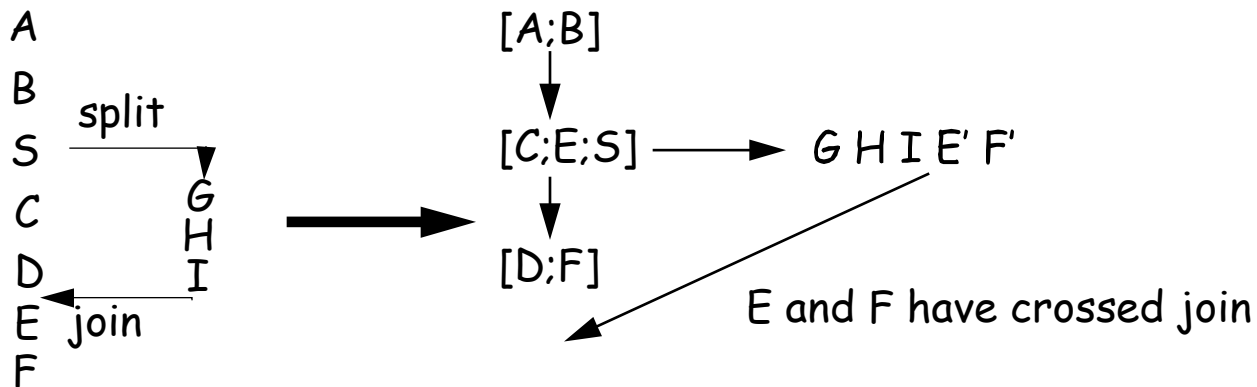
- ASSUME NO TRACE SCHEDULING. LOCAL SCHEDULING ONLY.

Clock	MemOp1	MemOp2	INTOp1	INTOp2	BROp
1	LD R4,0(R1)	LD R5,0(R2)	NOOP	NOOP	NOOP
2	NOOP	NOOP	NOOP	NOOP	NOOP
3	NOOP	NOOP	NOOP	NOOP	BEQ R5,R4,CLK6
4	NOOP	NOOP	ADDI R6,R4,5	NOOP	NOOP
5	NOOP	SD R6, 0(R3)	NOOP,5	NOOP	GOTO CLK 8
6	NOOP	NOOP	NOOP	ADDI R7,R5,2	NOOP
7	SD R7,0(R3)	NOOP	NOOP	NOOP	NOOP
8	NOOP	NOOP	ADD R8,R6,R4	NOOP	NOOP
9	NOOP	SD R8,0(R1)	NOOP	NOOP	NOOP

TRACE SCHEDULING

- ASSUME NOW THAT BEQ IS MOST LIKELY TO BE TAKEN
- THEN THE FIRST TRACE IS:

	LD R4,0(R1)	A
	LD R5,0(R2)	B
	BEQ R5,R4,LAB1	S (SPLIT)
LAB1:	ADDI R7,R5,2	C
	SD R7,0(R3)	D
LAB2:	ADD R8,R6,R4	E (JOIN)
	SD R8,0(R1)	F



TRACE SCHEDULING

- PROGRAM FOR FIRST TRACE

Clock	MemOp1	MemOp2	INTOp1	INTOp2	BROp
1	LD R4,0(R1)	LD R5,0(R2)	NOOP	NOOP	NOOP
2	NOOP	NOOP	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R5,2	ADD R8,R6,R4	BNE R5,R4,OUT
4	SD R7,0(R3)	SD R8,0(R1)	NOOP	NOOP	NOOP

- THE SECOND TRACE WITH COMPENSATION CODE IS:

```

ADDI R6,R4,5
SD  R6,0(R3)
ADD  R8,R6,R4    /moved up across join
SD  R8,0(R1)     /moved up across join
    
```

- THE OVERALL CODE BECOMES

Clock	MemOp1	MemOp2	INTOp1	INTOp2	BROp
1	LD R4,0(R1)	LD R5,0(R2)	NOOP	NOOP	NOOP
2	NOOP	NOOP	NOOP	NOOP	NOOP
3	NOOP	NOOP	ADDI R7,R5,2	ADD R8,R6,R4	BNE R5,R4,CLK5
4	SD R7,0(R3)	SD R8,0(R1)	NOOP	NOOP	GOTO CLK 8
5	NOOP	NOOP	ADDI R6,R4,5	NOOP	NOOP
6	NOOP	SD R6,0(R3)	NOOP	ADD R8,R6,R4	NOOP
7	NOOP	SD R8,0(R1)	NOOP	NOOP	NOOP

- SO, WHEN THE BRANCH PREDICTION IS CORRECT WE CAN EXECUTE THE CODE IN 4 CLOCKS. WITHOUT TRACE SCHEDULING THE NUMBER OF CLOCKS IS 7
- WHEN THE BRANCH PREDICTION IS NOT CORRECT WE EXECUTE THE CODE IN 6 CLOCKS WITHOUT TRACE SCHEDULING THE NUMBER OF CLOCKS IS 7

VLIW--HARDWARE SUPPORT

- **PREDICATED INSTRUCTIONS**
 - USED EXTENSIVELY IN INTEL IA-64
- **HARDWARE SUPPORT FOR COMPILER SPECULATION**
 - SPECULATIVE VALUES
 - EXCEPTIONS

CONDITIONAL/PREDICATED INSTRUCTIONS

- EXECUTED ONLY IF A CONDITION IS SATISFIED
- CONTROL DEPENDENCIES ARE CONVERTED INTO DATA DEPENDENCIES
- EXAMPLE:

NORMAL CODE	PREDICATED
BNEZ R1,L	CMOVZ R2,R3,R1
MOV R2,R3	

L:
- USEFUL FOR SHORT IF-THEN STATEMENTS
- MORE COMPLEX INSTRUCTIONS MAY SLOW DOWN CLOCK
- NEVERTHELESS, IN SOME ARCHITECTURES, ALL INSTRUCTIONS ARE PREDICATED (eg INTEL IA-64--ITANIUM)
- EXCEPTIONS!: ENABLE THEM ONLY IF THE CONDITION IS MET
- LIMITED TO TWO WAY CONDITIONALS

COMPILER SPECULATION WITH HARDWARE SUPPORT

- THE COMPILER MOVES INSTRUCTIONS UP THROUGH BRANCHES SO THAT THEY CAN BE EXECUTED BEFORE THE BRANCH CONDITION IS KNOWN
- ADVANTAGE" CREATES LONGER SCHEDULABLE CODE SEQUENCES (BIGGER BB) AND MORE ILP
- MAY NOT AFFECT EXCEPTION BEHAVIOR
- NEVER MOVE A STORE UP ACROSS A BRANCH
 - NOT A REAL PROBLEM SINCE WE WANT TO MOVE STORES DOWN, NOT UP

PRESERVING EXCEPTIONS

- POISON BITS

LW R1, 0(R3)	LW R1,0(R3)
BNEZ R1,L1	LW.s R14,0(R2)
LW R1, 0(R2)	BNEZ R1,L1
J L2	CHECK.s R14, repair
L1: ADD R1,R1,#4	J L2
L2: SW 0(R3),R1	L1 ADD R14,R1,#4
	L2 SW 0(R3),R14

- THE SPECULATIVE INSTRUCTION LW.s DOES NOT RAISE EXCEPTIONS
- RATHER, ON AN EXCEPTION, IT POISONS THE DESTINATION REGISTER AND FILLS IT WITH AN EXCEPTION CONDITION
- CHECK.s RAISE THE EXCEPTION IF ANY. THEN EXECUTES PATCH-UP CODE repair
- CORRECT EXCEPTION MODEL

- APPROXIMATE EXCEPTION MODEL WITH POISON BIT

LW R1, 0(R3)	LW R1,0(R3)
BNEZ R1,L1	LW.s R14,0(R2)
LW R1, 0(R2)	BEQZ R1,L3
J L2	ADD R14,R1,#4
L1: ADD R1,R1,#4	L3: SW 0(R3),R14
L2: SW 0(R3),R1	

- LW.s SETS THE POISON BIT FOR R14 IF EXCEPTION
- EXCEPTION IS DETECTED WHEN A NON-SPECULATIVE INSTRUCTION READS A POISONED REGISTER
- SPECULATIVE INSTRUCTIONS PROPAGATE POISONED VALUE TO DESTINATION
- A POISONED REGISTER IS RESET WHEN IT IS THE DESTINATION OF A NON-SPECULATIVE INSTRUCTION

SPECULATIVE MEMORY DISAMBIGUATION

- **MEMORY REFERENCE SPECULATION**

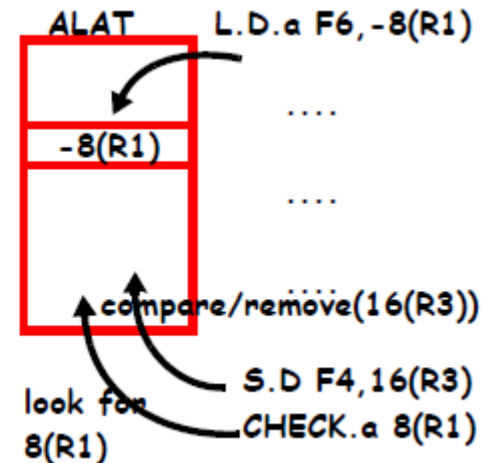
- TRY TO MOVE A LW ACROSS STORES W/O KNOWING IF THERE IS A CONFLICT
- PUT A SPECIAL INSTRUCTION (GUARDIAN) AT THE ORIGINAL LOCATION OF THE LW
- IF THE LOCATION IS NOT TOUCHED BETWEEN THE LW AND THE GUARDIAN, ALL IS FINE
- OTHERWISE, THE LW AND POSSIBLY ALL DEPENDENT INSTRUCTIONS ARE REPLAYED AT THE GUARDIAN

- **EXAMPLE**

Loop:

L.D	F0, 0(R1)
ADD.D	F4, F0, F2
S.D	F4, 0(R3)
L.D	F6, -8(R1)
ADD.D	F8, F6, F2
S.D	F8, -8(R3)
SUBI	R1, R1, #16
SUBI	R3, R3, #16
BNE	R1, R2, Loop

L.D	F0, 0(R1)
L.D.a	F6, -8(R1)
ADD.D	F4, F0, F2
ADD.D	F8, F6, F2
SUBI	R1, R1, #16
SUBI	R3, R3, #16
S.D	F4, 16(R3)
CHECK.a	8(R1), repair
S.D	F8, 8(R3)
BNE	R1, R2, Loop



- **ALAT: ADVANCED LOAD ADDRESS TABLE**

EPIC ARCHITECTURES

- EPIC: EXPLICITLY PARALLEL INSTRUCTION COMPUTER
- Intel IA-64 ISA
- registers:
 - 128 64-bit GP registers
 - 128 82-bit FP registers
 - 128 64-bit Application registers
 - 64 1-bit predicate register
 - 8 64-bit branch registers
- instructions: 5 execution unit slots: I-unit (Integer/multimedia), M-unit (integer or memory access), F-unit (Floating point) B-unit (branch) L+X (extended)
- Instruction group: sequence of consecutive instructions with no register dependencies among them. Could be scheduled all at once.
- A stop is inserted by the compiler between consecutive groups.
- Instructions are encoded in bundles(5-bit template+3 41-bit instructions)
- Template encodes stops and the execution unit for each instruction

IA-64 ISA

- **EXECUTION UNITS SLOTS:**
 - I-UNIT
 - A: INTEGER ALU (ADD, SUB, AND, OR, COMPARE)
 - I NON-ALU INTEGER (SHIFTS, BIT TEST, MOVE)
 - M-UNIT
 - A: INTEGER ALU (ADD, SUB, AND, OR, COMPARE)
 - M: MEMORY ACCESS (LD AND ST)
 - F-UNIT
 - FLOATING POINT
 - B-UNIT
 - BRANCHES
 - L+X
 - EXTENDED

IA-64 ISA BUNDLES AND GROUPS

```
( .mii
    add r1 = r2, r3
    sub r4 = r4, r5 ;;
    shr r7 = r4, r12 ;;
)
( .mmi
    ld8 r2 = [r1] ;;
    st8 [r1] = r23
    tbit p1,p2=r4,5
)
( .mbb
    ld8 r45 = [r55]
    (p3)br.call b1=func1
    (p4)br.cond Label1
)
( .mfi
    st4 [r45]=r6
    fmac f1=f2,f3
    add r3=r3,8 ;;
)
```


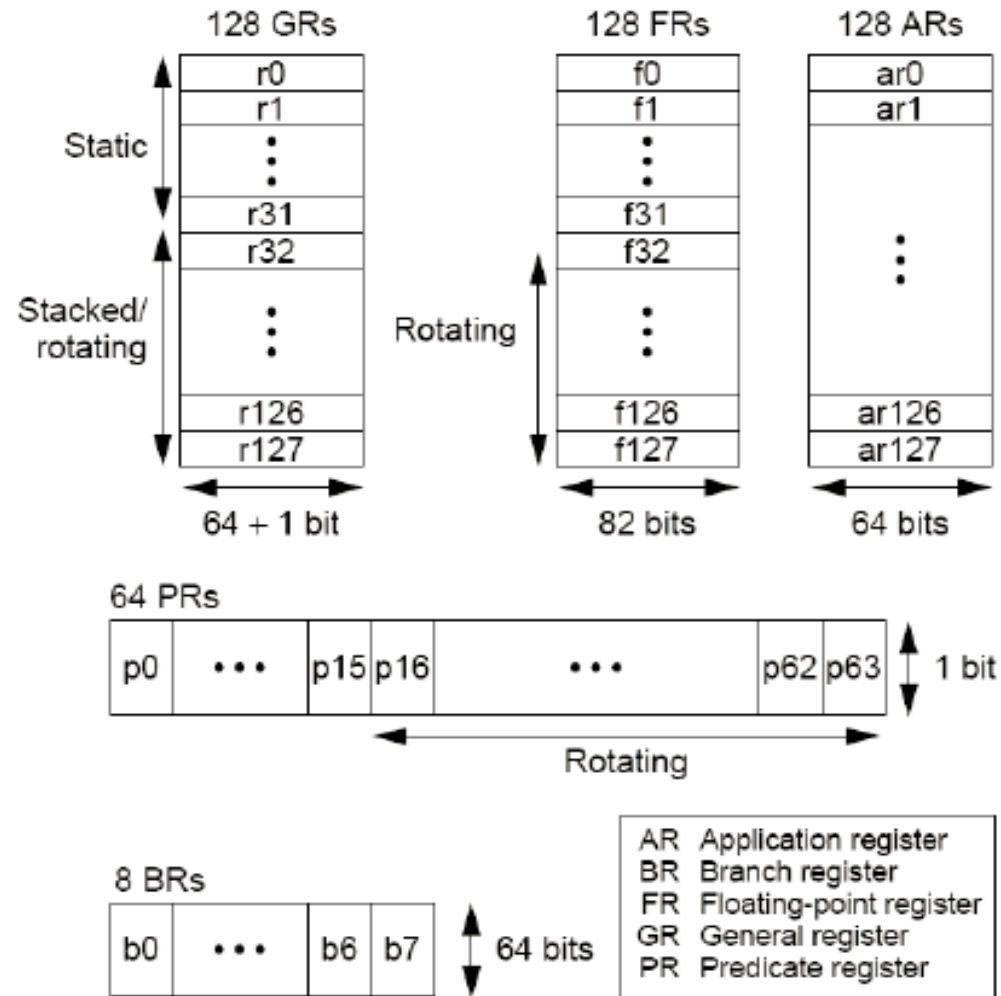


Figure 3. Example instruction groups.

IA-64 ISA REGISTERS



IA-64 ISA FORMATS



(a)



(b)

- **REGISTER STACKING: SUPPORT FOR PROCEDURE CALL**
 - REGISTER STACK ENGINE (RSE) SUPPORT REGISTER STACK OVERFLOW FOR PROCEDURE NESTING
- **REGISTER ROTATING: SUPPORT FOR SOFTWARE PIPELINING**
- **INSTRUCTION GROUP: NO REGISTER DEPENDENCY:**

IA-64 PREDICATION AND SPECULATION

- **PREDICATION:**
 - USE OF PREDICATE 1-BIT REGISTERS
 - P-REGISTERS ARE SET USING COMPARE AND TEST INSTRUCTIONS
 - EACH INSTRUCTION HAS 6-BITS TO SPECIFY A PREDICATE REGISTER
- **CONTROL SPECULATION**
 - DEFERRED EXCEPTIONS
 - POISON BITS ASSOCIATED WITH REGISTERS
 - NaTVaL (NOT A THING VALUE) PROPAGATE
 - GENERATED BY SPECULATIVE LOADS
 - ANY NON SPECULATIVE INSTRUCTION READING A NaTVaL TAKES A (DEFERRED) TRAP
 - CHK.S INSTRUCTIONS
- **MEMORY DISAMBIGUATION SPECULATION**
 - ADVANCED LOADS (LD.A)
 - LD.A INSERT ENTRY IN ALAT WITH LOAD ADDRESS
 - A SUBSEQUENT STORE WITH THE SAME @ REMOVES ENTRY
 - THE CHECK INSTRUCTION (CHK.A) LOOKS UP ALAT

IA-64

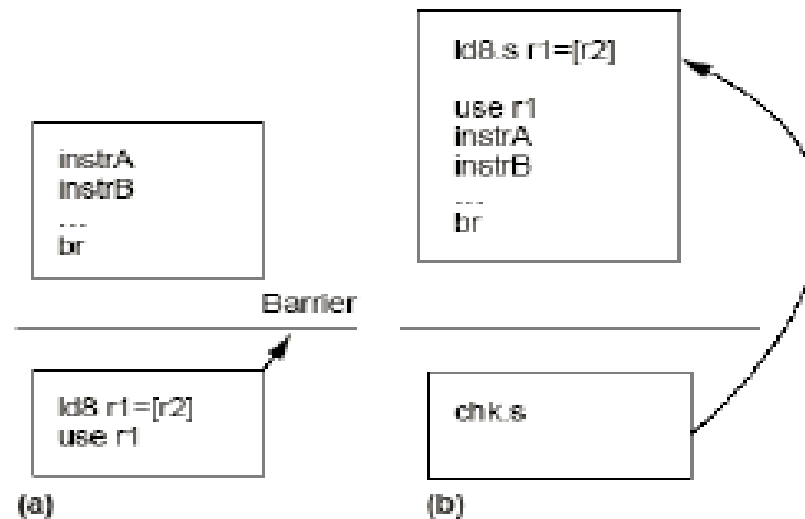
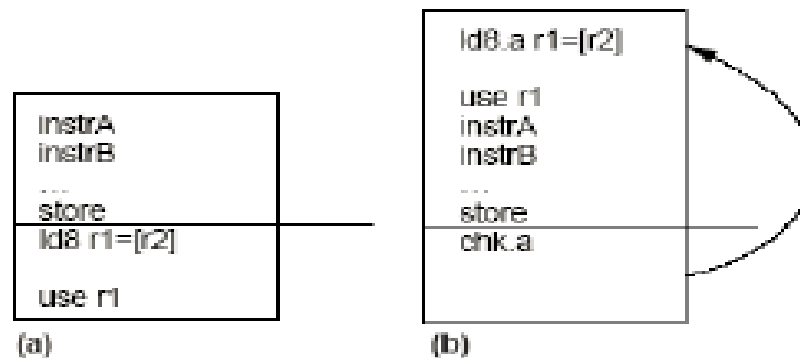


Figure 10. Comparing the scheduling of control speculative computations in traditional (a) and IA-64 (b) architectures. IA-64 allows elevation of loads and their uses above branches.



VECTOR PROCESSORS

- VECTOR PROCESSORS ARE ABLE TO EXECUTE INSTRUCTIONS ON ENTIRE VECTORS AND NOT JUST ON SCALARS
- INSTRUCTIONS ARE OF THE TYPE

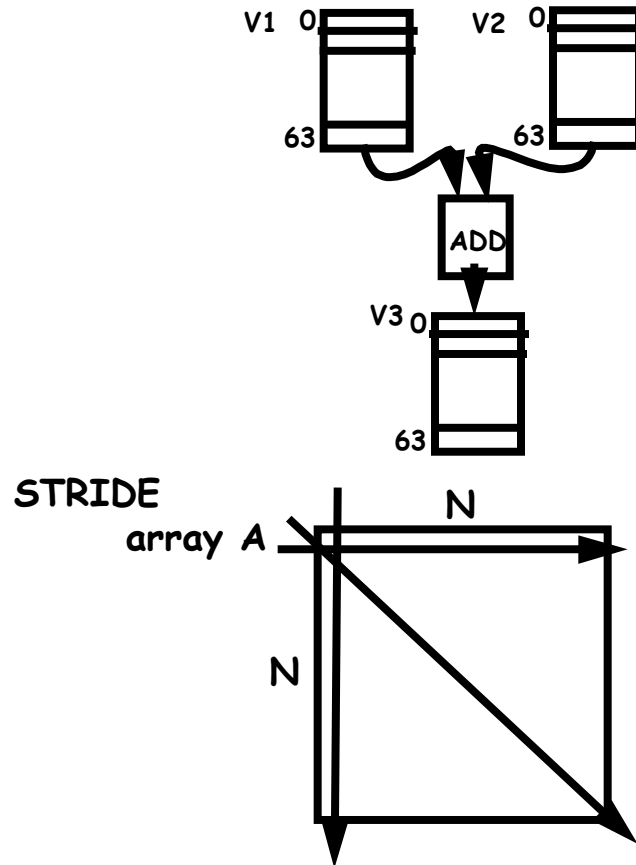
ADDV V1, V2, V3

- V1, V2, AND V3 ARE VECTORS OF SCALARS OF SAME TYPE AND SAME LENGTH
- THEY ARE SPECIFIED BY A BASE ADDRESS, A VECTOR LENGTH AND A STRIDE (MEMORY OPERAND), OR THEY CAN BE VECTOR REGISTERS
- THE ABOVE INSTRUCTION ADDs THE CORRESPONDING SCALAR ELEMENTS OF V2 AND V3 AND PUTS THE RESULTS IN CORRESPONDING ELEMENTS OF V1
 - $V1[i] \leftarrow V2[i] + V3[i]$, for all i 's
 - VECTOR LENGTH AND STRIDE MAY BE HELD IN SPECIAL CONTROL REGISTERS

**VECTOR MACHINES EXISTED WELL BEFORE SUPERSCALAR PROCESSORS
THEY ARE STILL BUILT FOR SUPERCOMPUTERS FOR ENGINEERING/SCIENTIFIC
COMPUTATION**

**BECAUSE OF MULTIMEDIA (STREAMING) APPLICATION THEY ARE REAPPEARING
IN COMMODITY MARKETS (DSP)**

VECTOR PROCESSORS



Assume ADD has 10 stages

The total execution time is:

$$T_{ex} = 10 + 63 = 9 + 64$$

In general: $T_{ex} = T_{start} + N$

- AFTER VECTOR STARTUP TIME (THE TIME TO GET THE FIRST RESULT), RESULTS ARE COMPUTED ONE PER CLOCK

$$T_{vector} = VECTOR_LENGTH + STARTUP\ TIME$$

LOAD/STORE VECTOR ARCHITECTURE

- **VECTOR REGISTERS + SCALAR REGISTERS**
 - EACH VECTOR REGISTER CAN HOLD CONSECUTIVE COMPONENTS OF VECTORS FETCHED FROM MEMORY
 - e.g. 8 VECTOR REGISTERS OF 64 COMPONENTS EACH
 - 1 READ AND 1 WRITE PORT PER VECTOR REGISTER CONNECTED TO FUNCTIONAL UNITS THROUGH CROSSBARS
- **VECTOR FUNCTIONAL UNITS ARE DEEPLY PIPELINED AND SPECIALIZED**
 - ACT ON OPERANDS IN VECTOR REGISTERS
- **EXAMPLE:** $Y = a * X + Y$

```
L.V      V1,0(R1),R6 /load vector X with base 0(R1) and stride (R6)
MUL.V    V2,V1, F0    /multiply X by scalar in F0
L.V      V3,0(R2),R6  /load vector Y
ADD.V    V4,V,V3      /add the vectors
S.V      0(R2),V4,R6  /store vector Y
```

LOOP STRIP-MINING:

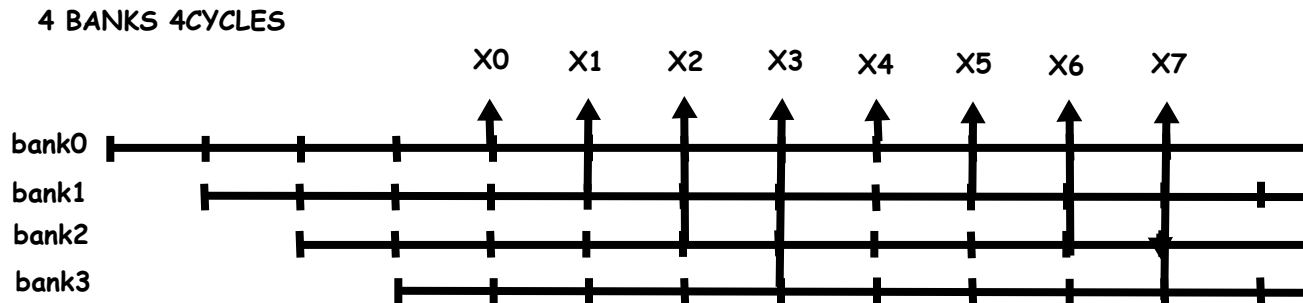
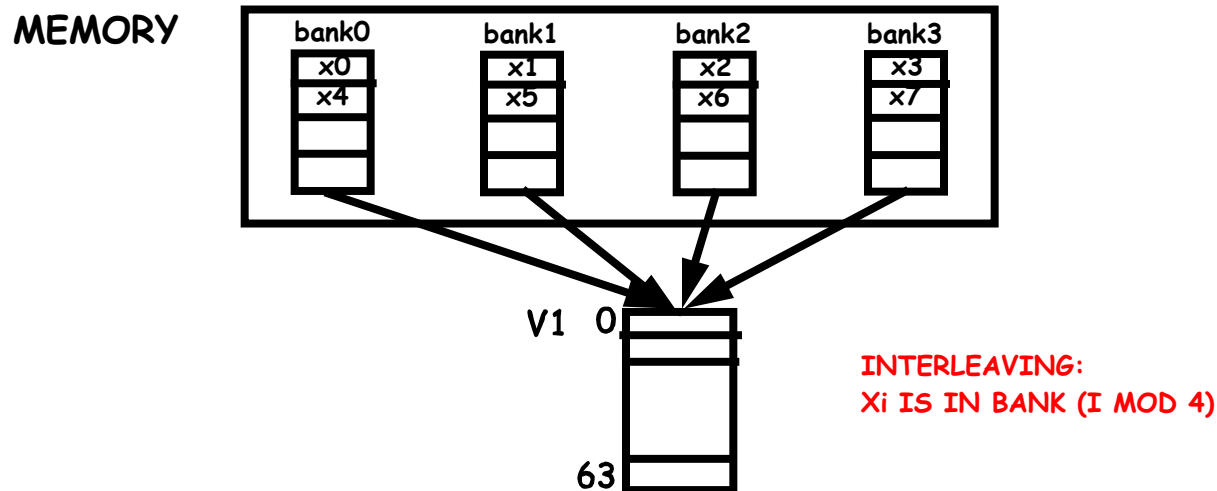
```
LOOP:    L.V      V1, 0(R1),R6    /load slice of vector X
          MUL.V    V2, V1,F0      /multiply X by scalar in F0
          L.V      V3, 0(R2),R6    /load slice of vector Y
          ADD.V    V4, V2,V3      /add the vector slices
          S.V      0(R2),V4,R6     /store slice of vector Y
          SUBI     R1, R1, #512    /assume that machine is byte addressable
          SUBI     R2, R2, #512
          BNEZ     R1, LOOP
```

VECTOR--MEMORY SYSTEM

- **ACCESS PATTERN TO MEMORY IS KNOWN AT DECODE TIME FOR THE ENTIRE VECTORS**
 - MEMORY IS INTERLEAVED, NO NEED FOR CACHES
 - VECTOR LOAD/STORE UNITS FROM MEMORY TO REGISTERS
- **LOAD STORE UNITS CAN BE SEEN AS PIPELINES**
 - THE STARTUP TIME IS THE TIME TO GET THE FIRST COMPONENT
 - BANKS ARE STARTED ONE AFTER THE OTHER
 - IF THE NUMBER OF BANKS IS GREATER THAN THE MEMORY CYCLE TIME, WE HAVE NO CONFLICTS, AND RESULTS COME OUT ONE PER CLOCK
 - STARTUP TIME IS MUCH LONGER THAN FOR FUNCTIONAL UNITS
 - VECTOR ACCESSED WITH A STRIDE ARE STORED IN CONSECUTIVE LOCATION OF VECTOR REGISTER

VECTOR--MEMORY ORGANIZATION

- HEAVILY INTERLEAVED (HUNDREDS OF MEMORY MODULES)
- SIMPLE EXAMPLE

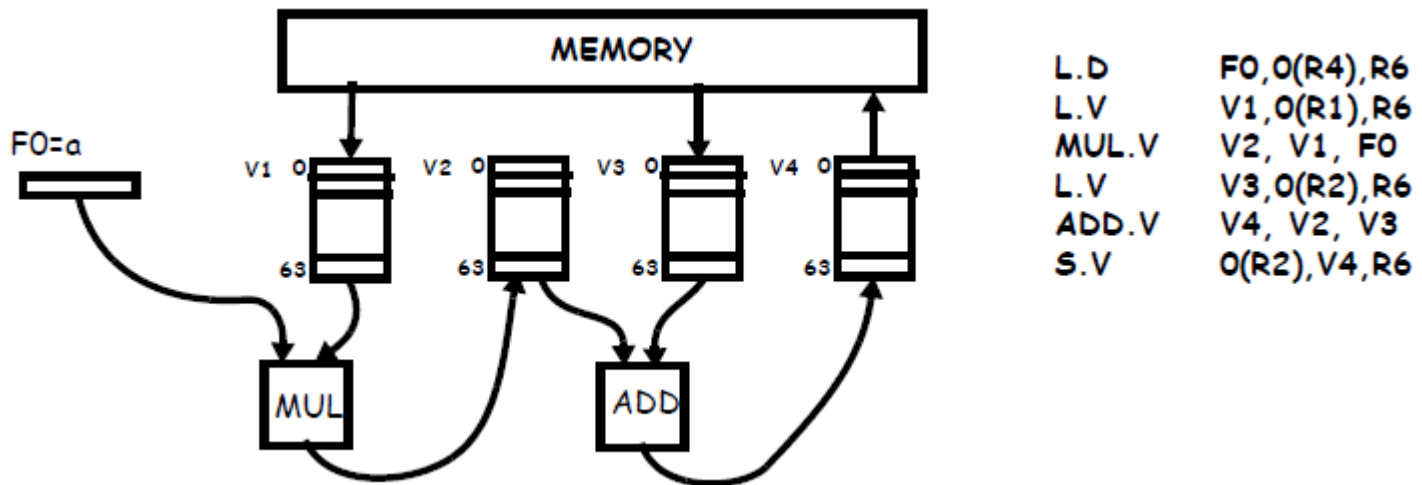


$$T_{\text{load}} = \text{VECTOR_LENGTH} + \text{STARTUP TIME (TIMETOGETX0 - 1)}$$

CHAINING AND PARALLEL EXECUTION

- INDEPENDENT VECTOR INSTRUCTIONS CAN BE EXECUTED IN PARALLEL PLUS CAN BE CHAINED
- CONSIDER THE CODE:

$$Y = a \times X + Y$$
- EXECUTION TIME (ONE OP AT A TIME): startup(load) + vector_length + startup(multv) + vector_length + startup(load) + vector length+ startup(addv) + vector_length + startup(store) + vector_length = startup + vector_lengthx5
- EXECUTION TIME (CHAINING+PARALLEL):
 startup(load) + startup(multv) + startup(addv) + startup(store) + vector_length
 (the two loads execute in parallel)



SPECIAL MECHANISMS

- FOR CONDITIONAL STATEMENTS
- CONSIDER THE CODE

```
do 100 i = 1, 64
    if (A(i).ne.0) then
        A(i) = A(i) - B(i)
    endif
100 continue
```

- USE A BIT-VECTOR MASK REGISTER VM

```
SNEVS      V1, F0 /set VM(i) if V1(i) != F0 (A is in V1, 0 IN F0)
SUB.V      V1, V1, V2 /subtract under Vector mask
```

VECTOR MASK HAS SAME FUNCTION AS PREDICATE REGISTERS IN EPIC

- **SCATTER/GATHER**

- MANY SCIENTIFIC COMPUTATIONS USE SPARSE MATRICES

- MOST COMPONENTS ARE 0
- BUT THE PATTERN IS NOT REGULAR
- COMPRESS A SPARSE INTO VECTORS OF NON-ZERO ELEMENT

$A[1:1000] \Rightarrow A^*[1:9], K[1:9]$

- A^* : NONZERO ELEMENTS OF A; K: INDEXES OF NONZERO ELEMENTS OF A

- CONSIDER THE FOLLOWING CODE:

```
do 100 i = 1, n
100 A(K(i)) = A(K(i)) + C(M(I))
```

MEMORY SCATTER/GATHER

- **MEMORY LOCATIONS OF VECTOR COMPONENTS ARE SPREAD OUT IN MEMORY**
- **USE SCATTER AND GATHER INSTRUCTIONS**
 - **GATHER** IS A LOAD INSTRUCTION LOADING THE COMPONENTS AT INDEXED ADDRESSES INTO CONSECUTIVE V-REGISTER LOCATIONS
 - **SCATTER** IS A STORE INSTRUCTION FROM V-REGISTER TO INDEXED ADDRESSES

L.V Vk,0(R1),R6 /load vector K in REGISTER Vk

LI.V $V_a, V_k, 0(R2)$ /load indexed from $A(K(i))$: gather

<work on V_a , put result in V_b >

SI.V 0(R2), V_k, V_b /store indexed to A(K(i)): scatter