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

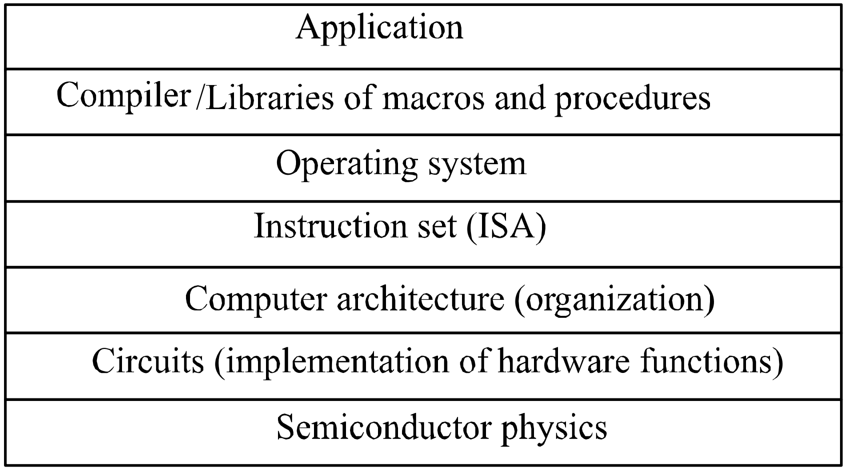
RISC vs. CISC

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 MULTPIPLE 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	reg[R4] <- reg[R4] +reg[R3]
IMMEDIATE	ADD R4, #3	reg[R4] <- reg[R4] + 3
DISPLACEMENT	ADD R4, 100(R1)	reg[R4] <- reg[R4] + Mem[100 + reg[R1]]
REGISTER INDIRECT	ADD R4, (R1)	reg[R4] <- reg[R4] + Mem[reg[R1]]
INDEXED	ADD R3, (R1+R2)	reg[R3] <- reg[R3] + Mem[reg[R1] + reg[R2]]
DIRECT OR ABSOLUTE	ADD R1, (1001)	reg[R1] <- reg[R1] + Mem[1001]
MEMORY INDIRECT	ADD R1, @R3	reg[R1] <- reg[R1] + Mem[Mem[Reg[3]]]
POST INCREMENT	ADD R1, (R2)+	ADD R1, (R2) then R2 <- R2+d
PREDECREMENT	ADD R1, -(R2)	R2 <- R2-d then ADD R1, (R2)
PC-RELATIVE	BEZ R1, 100	if R1==0, PC <- 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<- 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)

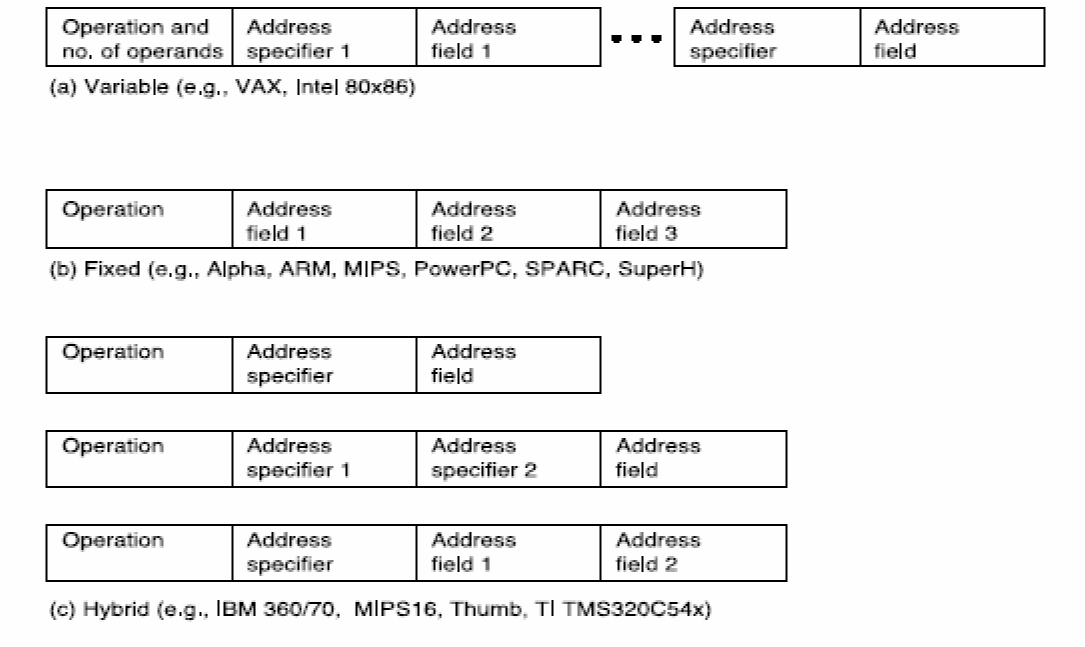
```
      LW R1,A
      LW R2,B
      ADD R3,R1,R2
      SW R3,C
```

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



Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier	Address field

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

Operation	Address field 1	Address field 2	Address field 3

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

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

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

INTEL iAPX 432!!

RISC vs. CISC

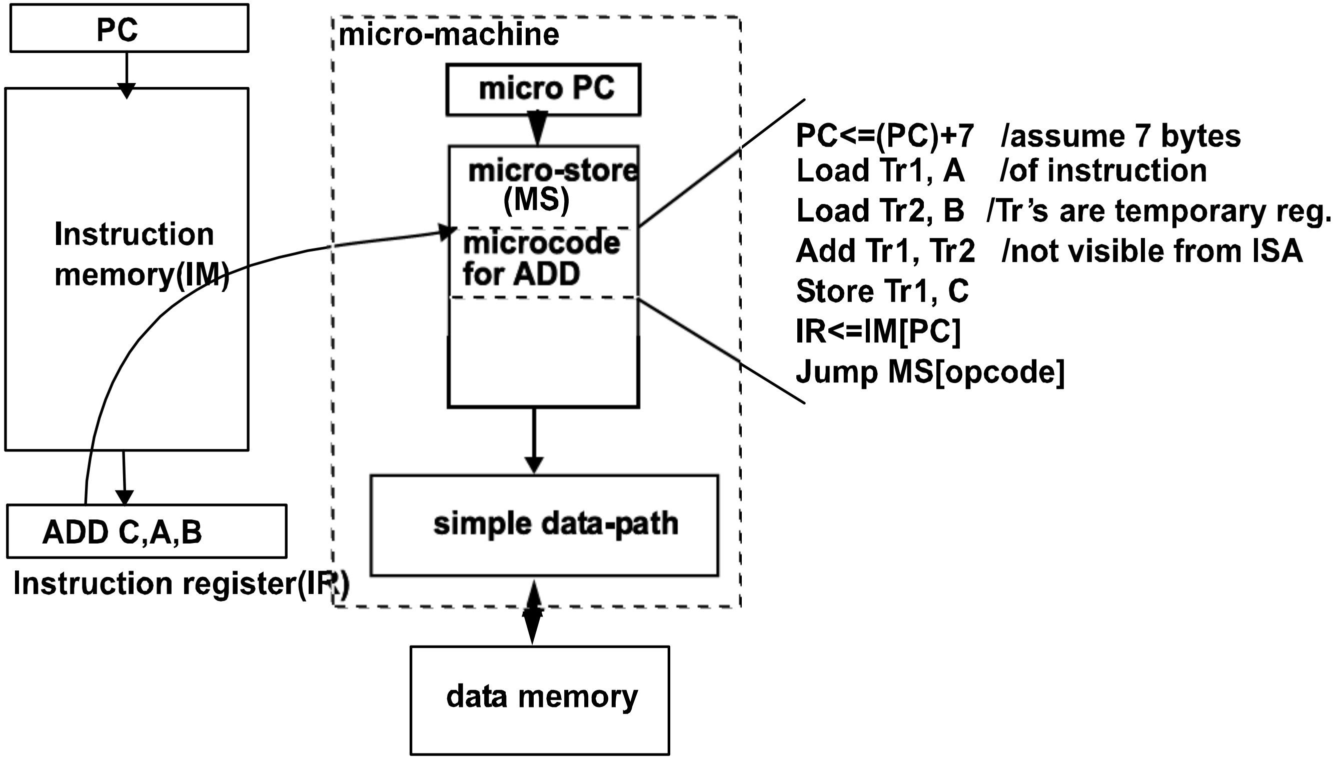
COMPLEX vs REDUCED (i.e., SIMPLE) INSTRUCTION SET COMPUTERS

NOT DIRECTLY RELATED TO IMPLEMENTATION

COMPLEX RISC IMPLEMENTATIONS

SIMPLE CISC IMPLEMENTATIONS

SIMPLE IMPLEMENTATION OF CISC MACHINES USE MICROCODE



```
PC<=(PC)+7     /assume 7 bytes
Load Tr1, A    /of instruction
Load Tr2, B   /Tr's are temporary reg.
Add Tr1, Tr2   /not visible from ISA
Store Tr1, C
IR<=IM[PC]
Jump MS[opcode]
```

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<=MEM[(R2)+20]	for bytes, half-words
	SB, SH, SW, SD	SW R1,#20(R2)	MEM[(R2)+20]<=(R1)	words, and double words
	L.S, L.D	L.S F0,#20(R2)	F0<=MEM[(R2)+20]	single/double float load
	S.S, S.D	S.S F0,#20(R2)	MEM[(R2)+20]<=(F0)	single/double float store
ALU operations	ADD, SUB, ADDU, SUBU	ADD R1,R2,R3	R1<=(R2)+(R3)	add/sub signed or unsigned
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1,R2,#3	R1<=(R2)+3	add/sub immediate signed or unsigned
	AND, OR, XOR,	AND R1,R2,R3	R1<=(R2).AND.(R3)	bitwise logical AND, OR, XOR
	ANDI, ORI, XORI,	ANDI R1,R2,#4	R1<=(R2).ANDI.4	bitwise AND, OR, XOR immediate
	SLT, SLTU	SLT R1,R2,R3	R1<=1 if R2<R3 else R1<=0	test on R2,R3 outcome in R1, signed or unsigned comparison
	SLTI, SLTUI	SLTI R1,R2,#4	R1<=1 if R2<4 else R1<=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,DIV.S	ADD.S F1,F2,F3	F1<=(F2)+(F3)	float arithmetic single precision
	ADD.D,SUB.D,MUL.D,DIV.D	ADD.D F0,F2,F4	F0<=(F2)+(F4)	float arithmetic double precision

INSTRUCTION FORMATS

31 26	25 21	20 16	15 0
opcode	Rs	Rt	displacement/immediate/offset

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

31 26	25 21	20 16	15 11	10 0
opcode	Rs	Rt	Rs	

ADD Rd, Rt, Rs

31 26	25 0
opcode	target

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

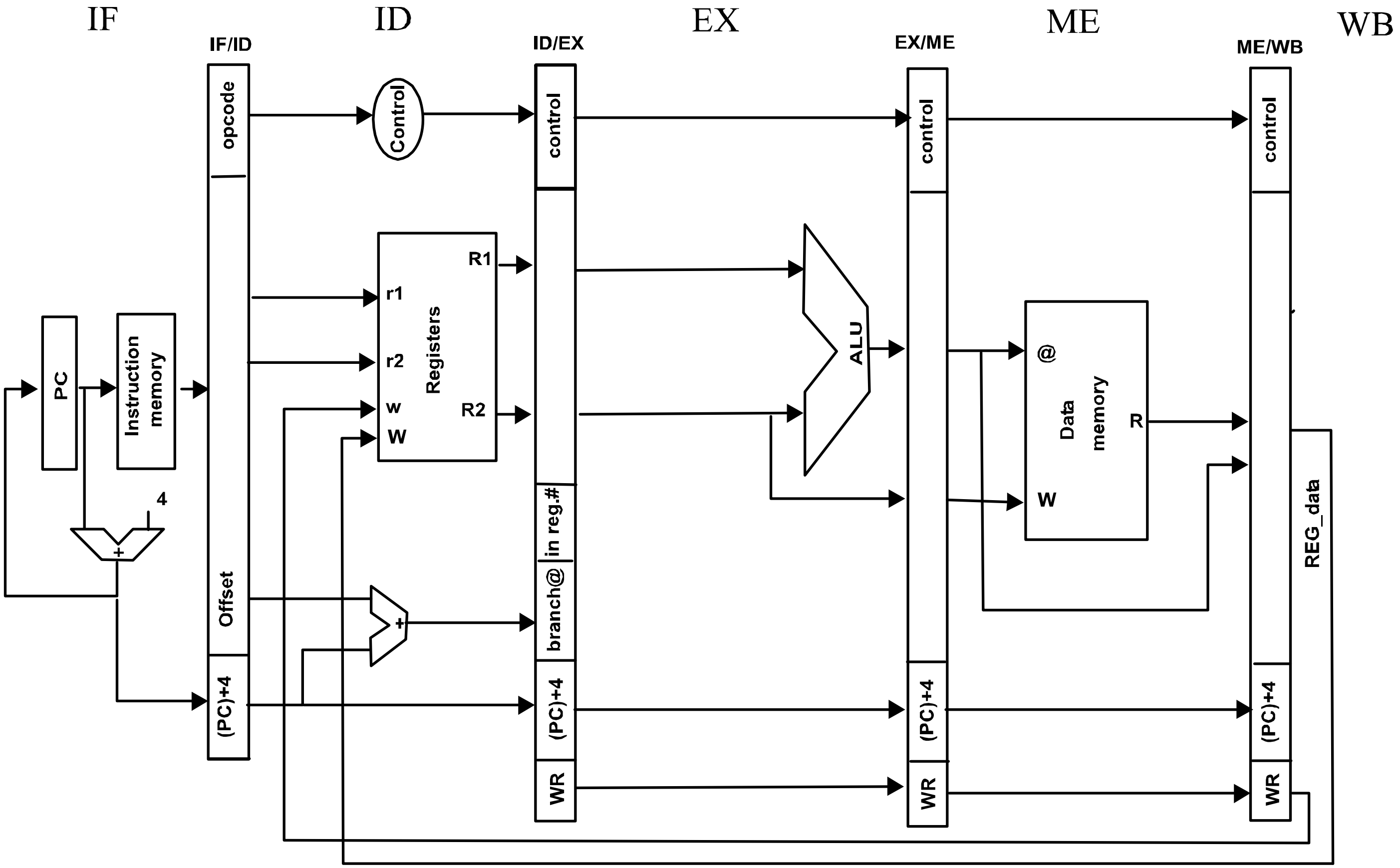
SOFTWARE PIPELINING

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 later.

5-STAGE PIPELINE



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 0N 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 FROMWB 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

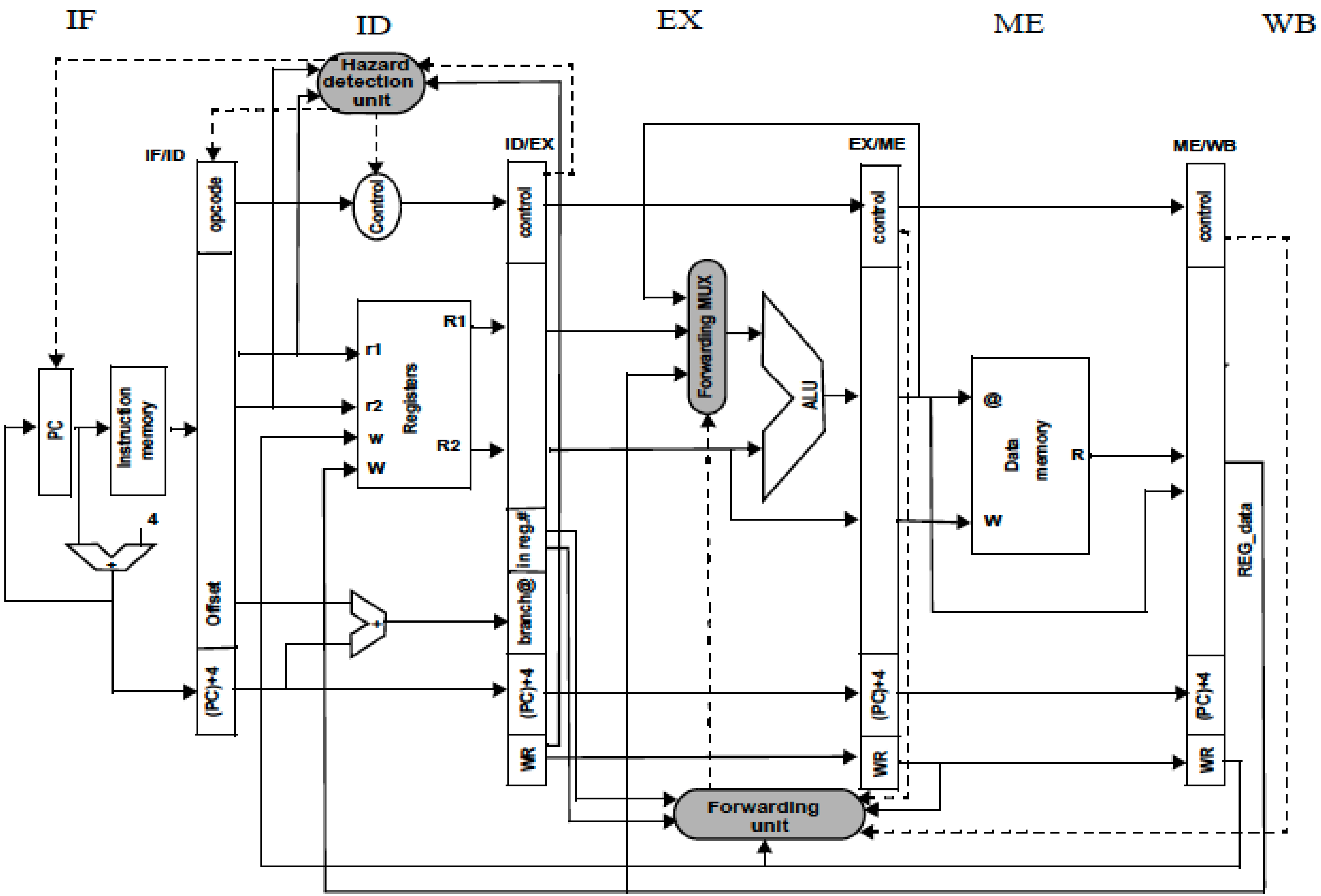
- DEPENDENCIES WITH PRIOR LOADs POSE SPECIAL COMPLICATIONS
- IN THIS CASE IT IS NOT POSSIBLE TO FORWARD THE VALUE FROM I1 TO I2 BECAUSE THE VALUE IS AVAILABLE ONLY IN CLOCK C5
  - HOWEVER, THE VALUE CAN BE FORWARDED FROM I1 TO I3
  - I2 MUST BE STALLED TO WAIT FOR THE VALUE OF I1

	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)

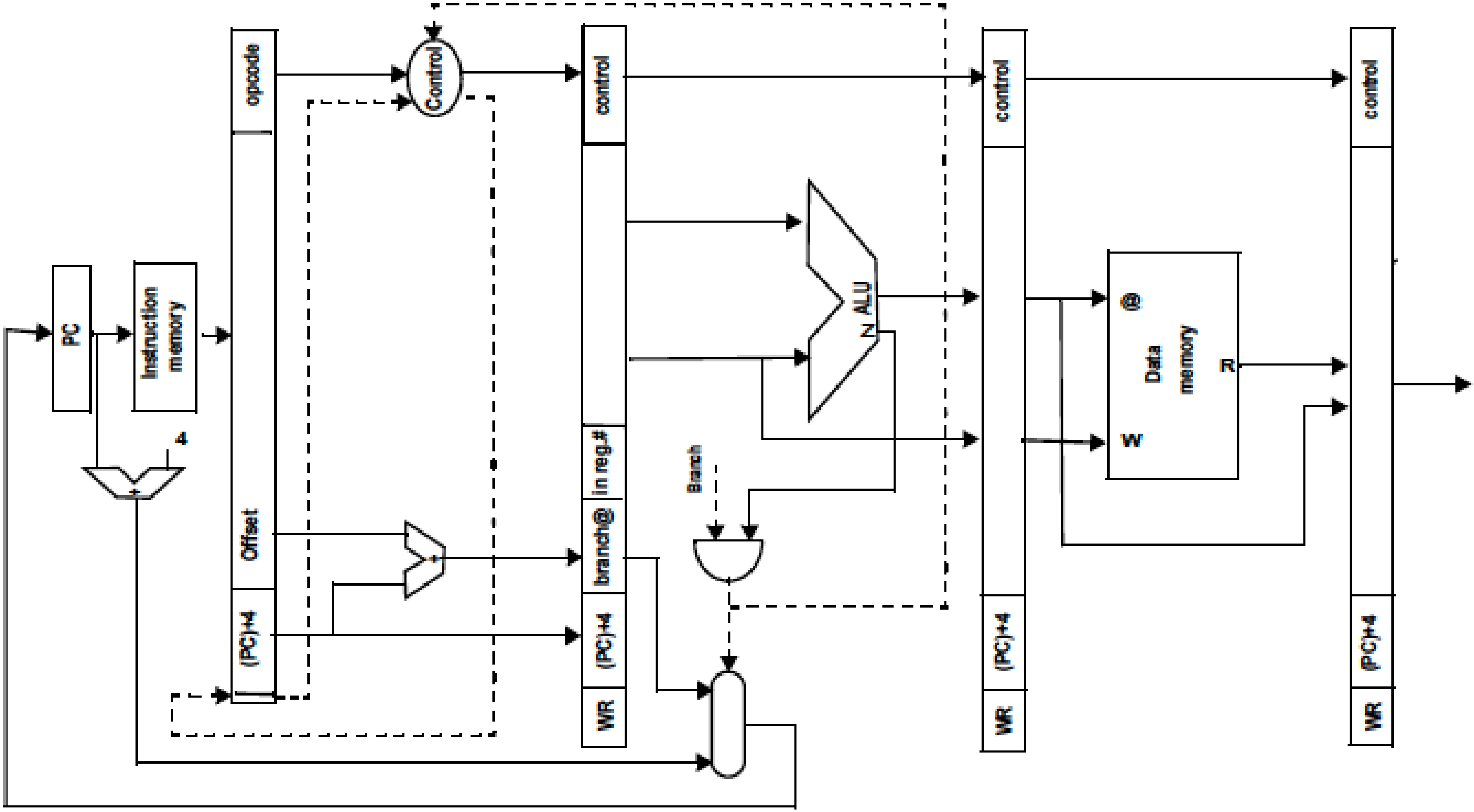
5 STAGE PIPELINE WITH FU AND HDU



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.

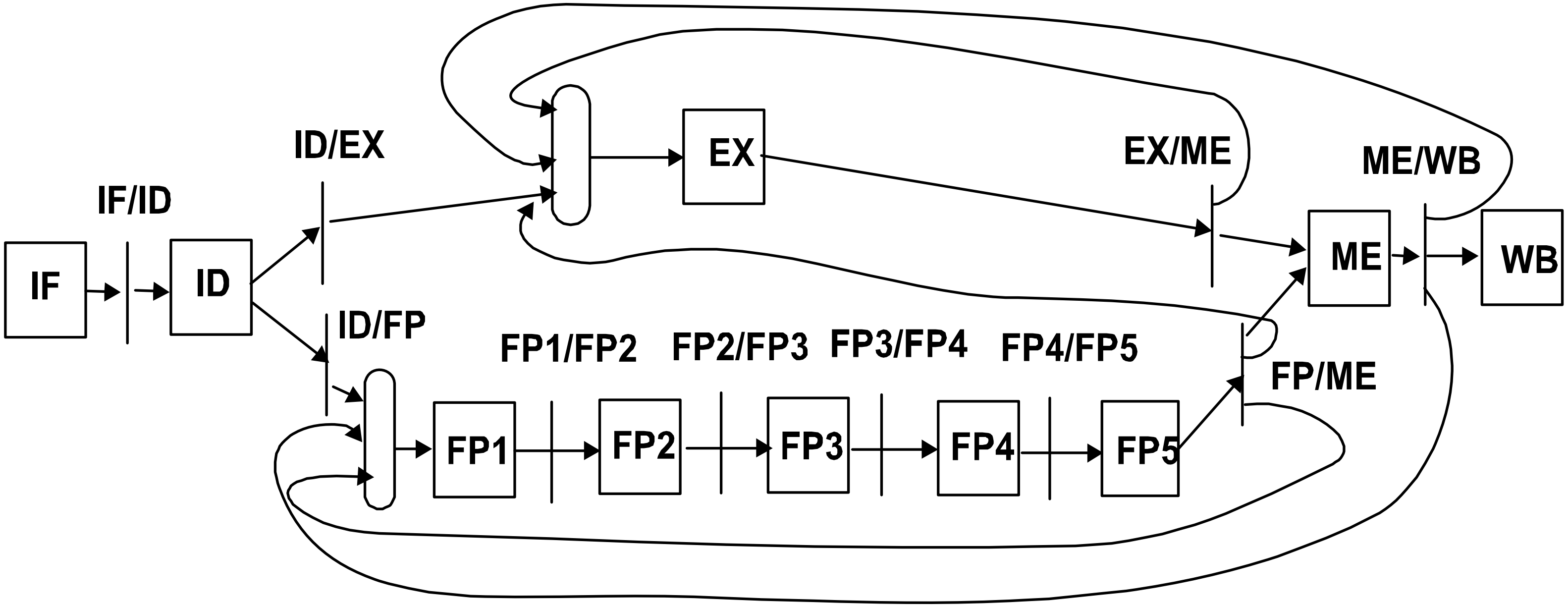
POSSIBLE PROBLEMS FOR PRECISE EXCEPTIONS IN CISC MACHINES

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 UNSTRUCTIONS
  - 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 CLOKCS

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 ID 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

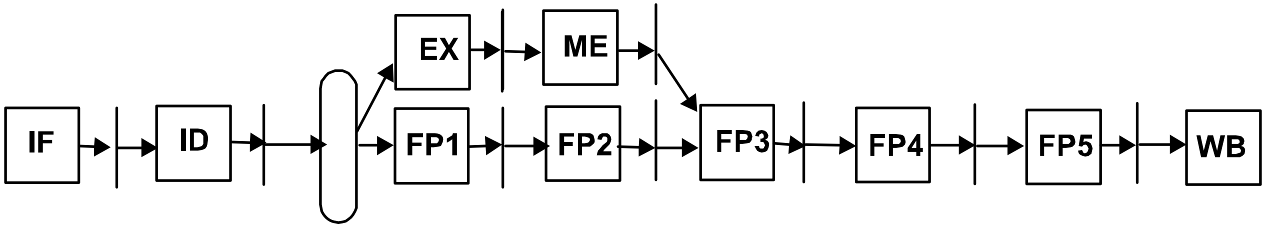
FORGET PRECISION ON EXCEPTIONS

- 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

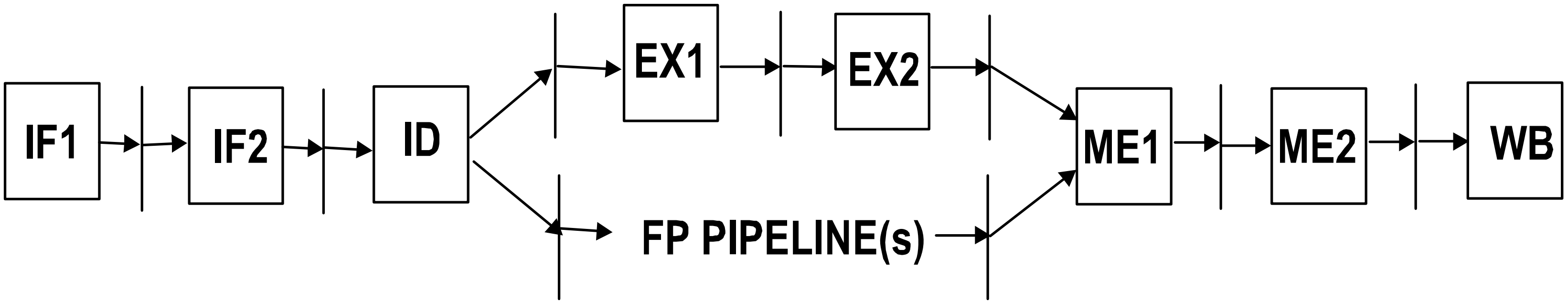
DEAL WITH EXCEPTIONS AS IF THEY WERE HAZARDS

- 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



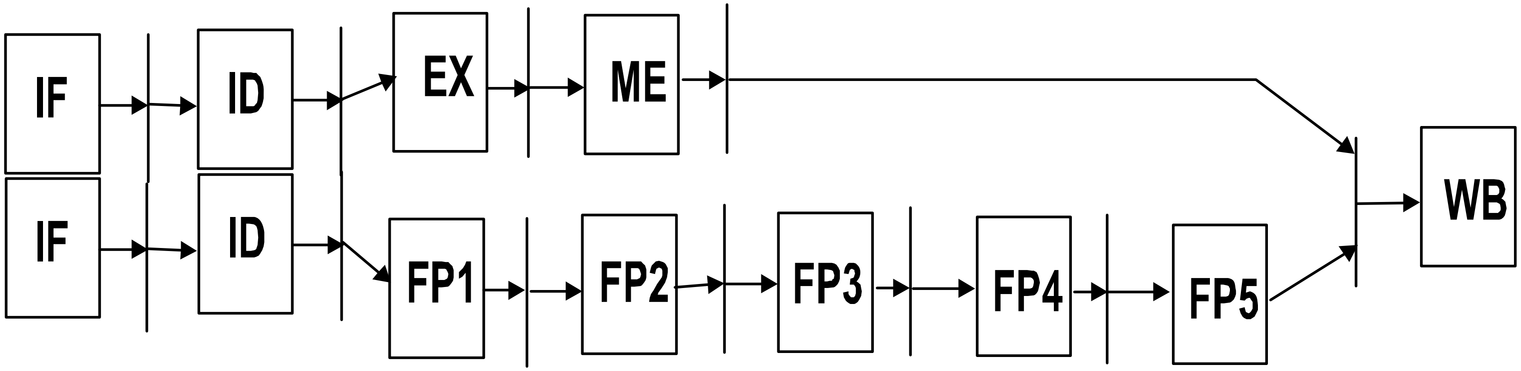
SOME STAGES IN THE 5-STAGE INTEGER PIPELINE ARE FURTHER PIPELINED

- 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

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

Dynamic Pipelines

- A dynamic pipeline is a pipeline where instructions can “step-out” of the pipeline until some condition is satisfied:
  - Sleep

- They can “step-in” to the pipeline when it is appropriate:
  - Wake-up
  - Select

- This allows dynamic scheduling of instructions.

Advantages of Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior

- It handles cases when dependences unknown at compile time
  - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve

- It allows code that compiled for one pipeline to run efficiently on a different pipeline

- It simplifies the compiler

- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling (later).

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

```
DIVD    F0,F2,F4
ADDD    F10,F0,F8
SUBD    F12,F8,F14
```

- Enables out-of-order execution and allows out-of-order completion (e.g., SUBD)
  - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (in-order issue)

- Will distinguish when an instruction *begins execution* and when it *completes execution*; between 2 times, the instruction is *in execution (or, in-flight).*

- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue*—Decode instructions, check for structural hazards
- *Read operands*—Wait until no data hazards, then read operands

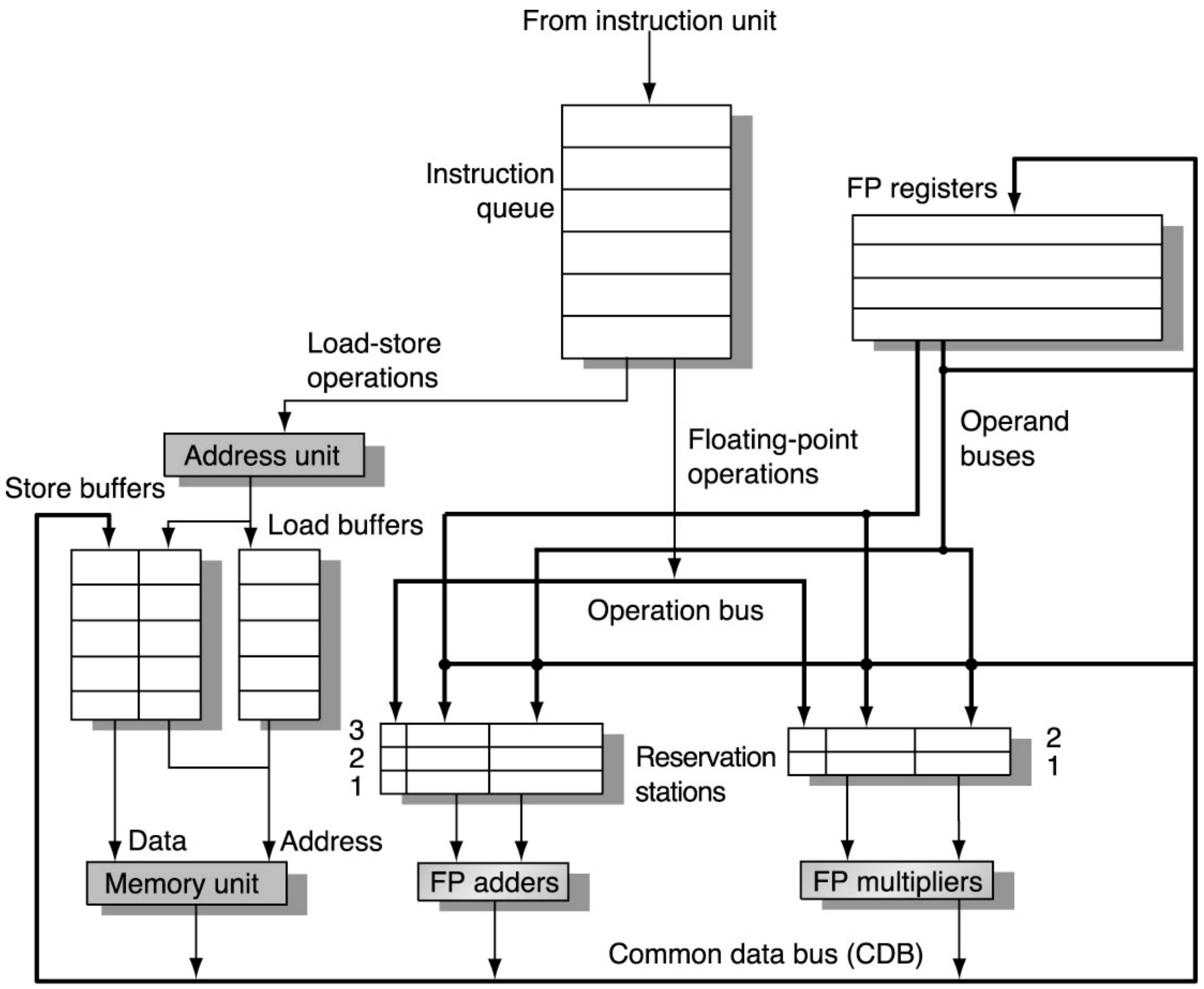
A Dynamic Algorithm: Tomasulo’s

- For IBM 360/91 (before caches!)
  - · Long memory latency
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers — renaming in hardware!
- Why Study 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, Pentium 4, AMD Opteron, Power 5, …

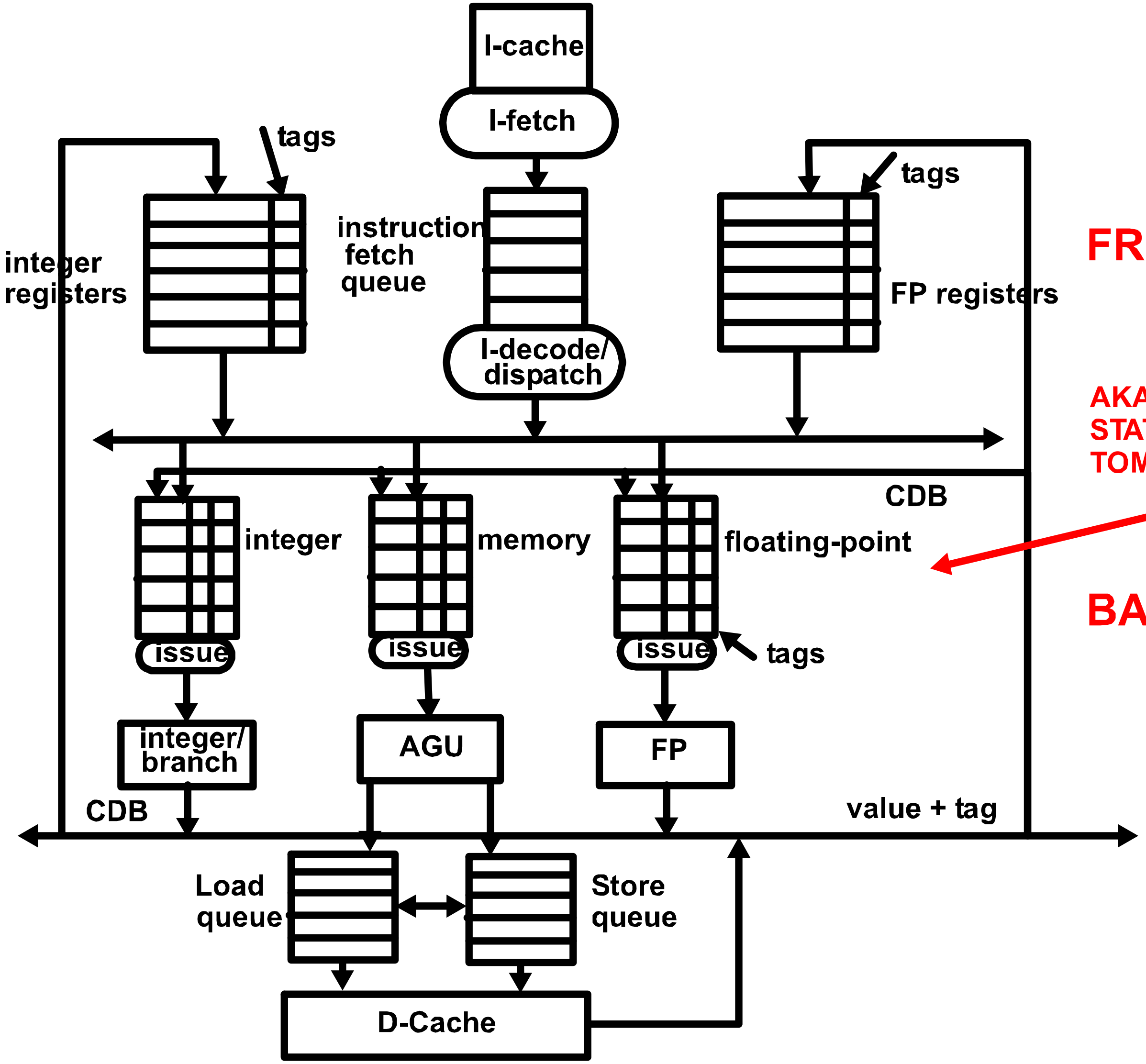
Tomasulo Algorithm

- Control & buffers distributed with Function Units (FU)
  - FU buffers called “reservation stations”; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - Renaming avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can’t
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
  - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue

Tomasulo Organization



TOMASULO ALGORITHM



FRONT-END

AKA "RESERVATION STATIONS" IN ORIGINAL TOMASULO

BACK-END

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: 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.

Why can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- Reservation stations
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall
- Other perspective: Tomasulo building data flow dependency graph on the fly

Tomasulo’s scheme offers 2 major advantages

1. Distribution of the hazard detection logic
   - distributed reservation stations and the CDB
   - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
   - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
2. Elimination of stalls for WAW and WAR hazards

Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264,
    IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units
    - high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ・ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

Reservation Station Components

Op—Operation to perform in the unit (e.g., + or –)

Qj, Qk—Reservation stations producing source registers

Vj, Vk—Value of Source operands

Rj, Rk—Flags indicating when Vj, Vk are ready

Busy—Indicates reservation station and FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

   If reservation station free, the instruction is issued & operands are sent (renames registers).

2. Execution—operate on operands (EX)

   When both operands ready then execute;
   if not ready, watch CDB for result

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting units;
   mark reservation station available.

Tomasulo Example Cycle 0

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULT	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU									

Tomasulo Example Cycle 1

Instruction status					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULT	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status										
Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

Tomasulo Example Cycle 2

Instruction status

					Execution	*Write*
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULT	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations

				S1	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*		Load2		Load1					

Tomasulo Example Cycle 3

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD		R(F4)	Load2	
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

Tomasulo Example Cycle 4

Instruction status					Execution	Write
Instruction	j	k		Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2		
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	Yes	SUBD	M(34+R2)			Load2
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD		R(F4)	Load2	
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(34+R2)	Add1				

Tomasulo Example Cycle 5

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	Yes	SUBD	M(34+R2)			Load2
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD		R(F4)	Load2	
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	Load2		M(34+R2)	Add1	Mult2			

Tomasulo Example Cycle 6

Instruction status					Execution	Write
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
2	Add1	Yes	SUBD	M(34+R2)	M(45+R3)		
0	Add2	Yes	ADDD		M(45+R3)	Add1	
	Add3	No					
10	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status										
Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	M(45+R3)		Add2	Add1	Mult2			

Tomasulo Example Cycle 7

Instruction status					Execution	Write
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
1	Add1	Yes	SUBD	M(34+R2)	M(45+R3)		
0	Add2	Yes	ADDD		M(45+R3)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status										
Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(45+R3)		Add2	Add1	Mult2			

Tomasulo Example Cycle 8

Instruction status

					Execution	*Write*
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	Yes	SUBD	M(34+R2)	M(45+R3)		
0	Add2	Yes	ADDD		M(45+R3)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(45+R3)		Add2	Add1	Mult2			

Tomasulo Example Cycle 9

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	Yes	ADDD	M()–M()	M(45+R3)		
	Add3	No					
7	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 10

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
2	Add2	Yes	ADDD	M()–M()	M(45+R3)		
	Add3	No					
6	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	FU	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 11

Instruction status

Instruction		j	k	Issue	Execution complete	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
1	Add2	Yes	ADDD	M()–M()	M(45+R3)		
	Add3	No					
5	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 12

Instruction status					*Execution*	*Write*
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	Yes	ADDD	M()–M()	M(45+R3)		
	Add3	No					
4	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(45+R3)		Add2	M()–M()	Mult2			

Tomasulo Example Cycle 13

Instruction status					*Execution*	*Write*
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 14

Instruction status

					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 15

Instruction status

Instruction		j	k	Issue	Execution	Write
					complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3		
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 16

Instruction status

					Execution	*Write*
Instruction		*j*	*k*	*Issue*	*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(45+R3)	R(F4)		
0	Mult2	Yes	DIVD		M(34+R2)	Mult1	

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	Mult1	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 17

Instruction status					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	17
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(34+R2)		

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
17	FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 18

Instruction status					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	17
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(34+R2)		

Register result status									
Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
18	FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 57

Instruction status					Execution	Write
Instruction	*j*	*k*	*Issue*		*complete*	*Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	17
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(34+R2)		

Register result status

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
57	*FU*	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 58

Instruction status

					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	17
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5	58	
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(34+R2)		

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
58	FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	Mult2			

Tomasulo Example Cycle 59

Instruction status

					Execution	Write
Instruction		j	k	Issue	complete	Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	5	6
MULT	F0	F2	F4	3	16	17
SUBD	F8	F6	F2	4	8	9
DIVD	F10	F0	F6	5	58	59
ADDD	F6	F8	F2	6	12	13

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
59	FU	M*F4	M(45+R3)		(M–M)+M()	M()–M()	M*F4/M			

Tomasulo Loop Example

```
Loop: LD          F0    0     R1
  MULTD     F4    F0    F2
  SD        F4    0     R1
  SUBI            R1    R1    #8
  BNEZ            R1    Loop
```

- Multiply takes 4 clocks
- Load have cache misses

Loop Example Cycle 0

Instruction status

					Execution	Write
Instruction	j	k	iteration	Issue	complete	Result
LD F0	0	R1	1			
MULT F4	F0	F2	1			
SD F4	0	R1	1			
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

	Busy	Address	Qi
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
0	**80**	*Qi*								

Loop Example Cycle 1

Instruction status

						Execution	Write
Instruction		j	k	iteration	Issue	complete	Result
LD	F0	0	R1	1	1		
MULT	F4	F0	F2	1			
SD	F4	0	R1	1			
LD	F0	0	R1	2			
MULT	F4	F0	F2	2			
SD	F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
1	80	Qi	Load1							

Loop Example Cycle 2

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1		
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1			
LD	F0	0	R1	2			
MULT	F4	F0	F2	2			
SD	F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
2	**80**	Qi	Load1		Mult1					

Loop Example Cycle 3

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
3	**80**	Qi	Load1		Mult1					

Loop Example Cycle 4

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2			
MULT F4	F0	F2	2			
SD F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
4	72	Qi	Load1		Mult1					

Loop Example Cycle 5

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result
LD	F0	0	R1	1	1		
MULT	F4	F0	F2	1	2		
SD	F4	0	R1	1	3		
LD	F0	0	R1	2			
MULT	F4	F0	F2	2			
SD	F4	0	R1	2			

	Busy	Address	Qi
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
5	72	Qi	Load1		Mult1						

Loop Example Cycle 6

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD F0	0	R1	1	1			Load1	Yes	80	
MULT F4	F0	F2	1	2			Load2	Yes	72	
SD F4	0	R1	1	3			Load3	No		Qi
LD F0	0	R1	2	6			Store1	Yes	80	Mult1
MULT F4	F0	F2	2				Store2	No		
SD F4	0	R1	2				Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
6	**72**	Qi	Load1		Mult1					

Loop Example Cycle 7

Instruction status

					Execution	Write				
Instruction	j	k	iteration	Issue	complete	Result		Busy	Address	
LD F0	0	R1	1	1			Load1	Yes	80	
MULT F4	F0	F2	1	2			Load2	Yes	72	
SD F4	0	R1	1	3			Load3	No		Qi
LD F0	0	R1	2	6			Store1	Yes	80	Mult1
MULT F4	F0	F2	2	7			Store2	No		
SD F4	0	R1	2				Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Qi	Load2		Mult2						

Loop Example Cycle 8

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1		
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6		
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
8	**72**	Qi	Load2		Mult2					

Loop Example Cycle 9

Instruction status						Execution	Write				
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address	
LD	F0	0	R1	1	1	9		Load1	Yes	80	
MULT	F4	F0	F2	1	2			Load2	Yes	72	
SD	F4	0	R1	1	3			Load3	No		Qi
LD	F0	0	R1	2	6			Store1	Yes	80	Mult1
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD	F4	0	R1	2	8			Store3	No		

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load1	
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
9	**64**	Qi	Load2		Mult2						

Loop Example Cycle 10

Instruction status					Execution	Write				
Instruction	j	k	iteration	Issue	complete	Result		Busy	Address	
LD F0	0	R1	1	1	9	10	Load1	No		
MULT F4	F0	F2	1	2			Load2	Yes	72	
SD F4	0	R1	1	3			Load3	No		Qi
LD F0	0	R1	2	6	10		Store1	Yes	80	Mult1
MULT F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD F4	0	R1	2	8			Store3	No		

Reservation Stations				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
4	Mult1	Yes	MULTD	M(80)	R(F2)		
0	Mult2	Yes	MULTD		R(F2)	Load2	

Code:

```
LD     F0     0   R1
MULT   F4     F0  F2
SD     F4     0   R1
SUBI   R1     R1  #8
BNEZ   R1     Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
10	64	Qi	Load2		Mult2					

Loop Example Cycle 11

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD F0	0	R1	1	1	9	10	Load1	No		
MULT F4	F0	F2	1	2			Load2	No		
SD F4	0	R1	1	3			Load3	Yes	64	Qi
LD F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1
MULT F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD F4	0	R1	2	8			Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
3	Mult1	Yes	MULTD	M(80)	R(F2)		
4	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:

```
LD     F0      0  R1
MULT   F4     F0  F2
SD     F4      0  R1
SUBI   R1     R1  #8
BNEZ   R1     Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
11	**64**	Qi			Mult2					

Loop Example Cycle 12

<u>Instruction status</u>

						Execution	Write				
Instruction		j	k	iteration	Issue	complete	Result		Busy	Address	
LD	F0	0	R1	1	1	9	10	Load1	No		
MULT	F4	F0	F2	1	2			Load2	No		
SD	F4	0	R1	1	3			Load3	Yes	64	Qi
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	Mult1
MULT	F4	F0	F2	2	7			Store2	Yes	72	Mult2
SD	F4	0	R1	2	8			Store3	No		

<u>Reservation Stations</u>

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
2	Mult1	Yes	MULTD	M(80)	R(F2)		
3	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

<u>Register result status</u>

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
12	**64**	Qi			Mult2					

Loop Example Cycle 13

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2		
SD F4	0	R1	1	3		
LD F0	0	R1	2	6	10	11
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
1	Mult1	Yes	MULTD	M(80)	R(F2)		
2	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
13	**64**	Qi			Mult2					

Loop Example Cycle 14

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2	14	
SD F4	0	R1	1	3		
LD F0	0	R1	2	6	10	11
MULT F4	F0	F2	2	7		
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD	M(80)	R(F2)		
1	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
14	**64**	Qi			Mult2						

Loop Example Cycle 15

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD	F0	0	R1	1	1	9	10	Load1	No		
MULT	F4	F0	F2	1	2	14	15	Load2	No		
SD	F4	0	R1	1	3			Load3	Yes	64	Qi
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F
MULT	F4	F0	F2	2	7	15		Store2	Yes	72	Mult2
SD	F4	0	R1	2	8			Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	No					
0	Mult2	Yes	MULTD	M(72)	R(F2)		

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
15	**64**	Qi			Mult2						

Loop Example Cycle 16

Instruction status

Instruction		j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD	F0	0	R1	1	1	9	10	Load1	No		
MULT	F4	F0	F2	1	2	14	15	Load2	No		
SD	F4	0	R1	1	3			Load3	Yes	64	Qi
LD	F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F
MULT	F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7
SD	F4	0	R1	2	8			Store3	No		

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
16	**64**	Qi			Mult1					

Loop Example Cycle 17

Instruction status					*Execution*	*Write*				
Instruction	*j*	*k*	*iteration*	*Issue*	*complete*	*Result*		Busy	Address	
LD F0	0	R1	1	1	9	10	Load1	No		
MULT F4	F0	F2	1	2	14	15	Load2	No		
SD F4	0	R1	1	3			Load3	Yes	64	Qi
LD F0	0	R1	2	6	10	11	Store1	Yes	80	M(80)*R(F
MULT F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7
SD F4	0	R1	2	8			Store3	Yes	64	Mult1

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

LD	F0	0	R1
MULT	F4	F0	F2
SD	F4	0	R1
SUBI	R1	R1	#8
BNEZ	R1	Loop	

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12 ...*	*F30*
17	**64**	*Qi*			Mult1					

Loop Example Cycle 18

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2	14	15
SD F4	0	R1	1	3	18	
LD F0	0	R1	2	6	10	11
MULT F4	F0	F2	2	7	15	16
SD F4	0	R1	2	8		

	Busy	Address	
Load1	No		
Load2	No		
Load3	Yes	64	Qi
Store1	Yes	80	M(80)*R(F
Store2	Yes	72	M(72)*R(7
Store3	Yes	64	Mult1

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
18	**56**	Qi			Mult1					

Loop Example Cycle 19

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result
LD F0	0	R1	1	1	9	10
MULT F4	F0	F2	1	2	14	15
SD F4	0	R1	1	3	18	19
LD F0	0	R1	2	6	10	11
MULT F4	F0	F2	2	7	15	16
SD F4	0	R1	2	8		

	Busy	Address	Qi
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	Yes	72	M(72)*R(7
Store3	Yes	64	Mult1

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

```
LD    F0    0  R1
MULT  F4   F0  F2
SD    F4    0  R1
SUBI  R1   R1  #8
BNEZ  R1   Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
19	**56**	Qi			Mult1					

Loop Example Cycle 20

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD F0	0	R1	1	1	9	10	Load1	No		
MULT F4	F0	F2	1	2	14	15	Load2	No		
SD F4	0	R1	1	3	18	19	Load3	Yes	64	Qi
LD F0	0	R1	2	6	10	11	Store1	No		
MULT F4	F0	F2	2	7	15	16	Store2	Yes	72	M(72)*R(7
SD F4	0	R1	2	8	20		Store3	Yes	64	Mult1

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

```
LD    F0     0  R1
MULT  F4    F0  F2
SD    F4     0  R1
SUBI  R1    R1  #8
BNEZ  R1    Loop
```

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
20	**56**	Qi			Mult1					

Loop Example Cycle 21

Instruction status

Instruction	j	k	iteration	Issue	Execution complete	Write Result		Busy	Address	
LD F0	0	R1	1	1	9	10	Load1	No		
MULT F4	F0	F2	1	2	14	15	Load2	No		
SD F4	0	R1	1	3	18	19	Load3	Yes	64	Qi
LD F0	0	R1	2	6	10	11	Store1	No		
MULT F4	F0	F2	2	7	15	16	Store2	No		
SD F4	0	R1	2	8	20	21	Store3	Yes	64	Mult1

Reservation Stations

				S1	S2	RS for j	RS for k
Time	Name	Busy	Op	Vj	Vk	Qj	Qk
0	Add1	No					
0	Add2	No					
0	Add3	No					
0	Mult1	Yes	MULTD		R(F2)	Load3	
0	Mult2	No					

Code:

LD F0 0 R1
MULT F4 F0 F2
SD F4 0 R1
SUBI R1 R1 #8
BNEZ R1 Loop

Register result status

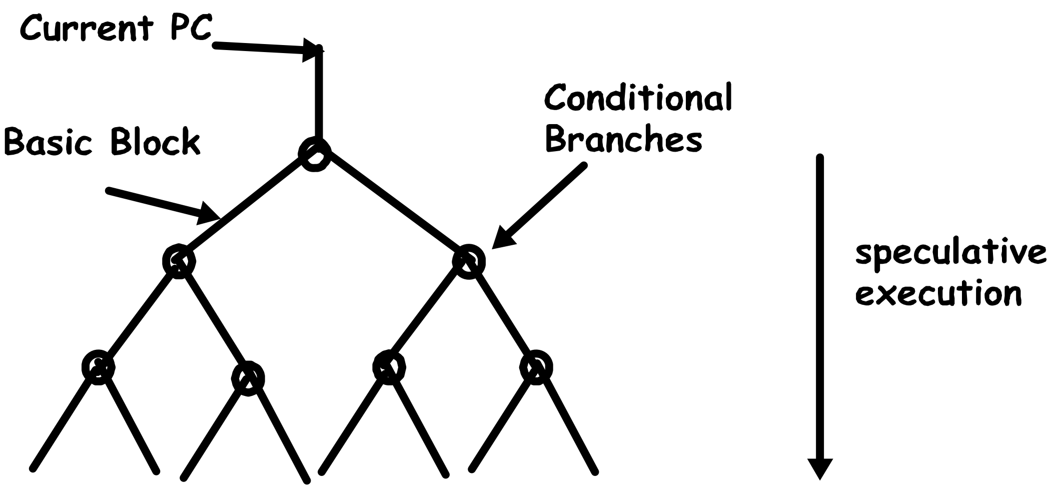
Clock	R1		F0	F2	F4	F6	F8	F10	F12 ...	F30
21	**56**	Qi			Mult1					

Tomasulo Summary

- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

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

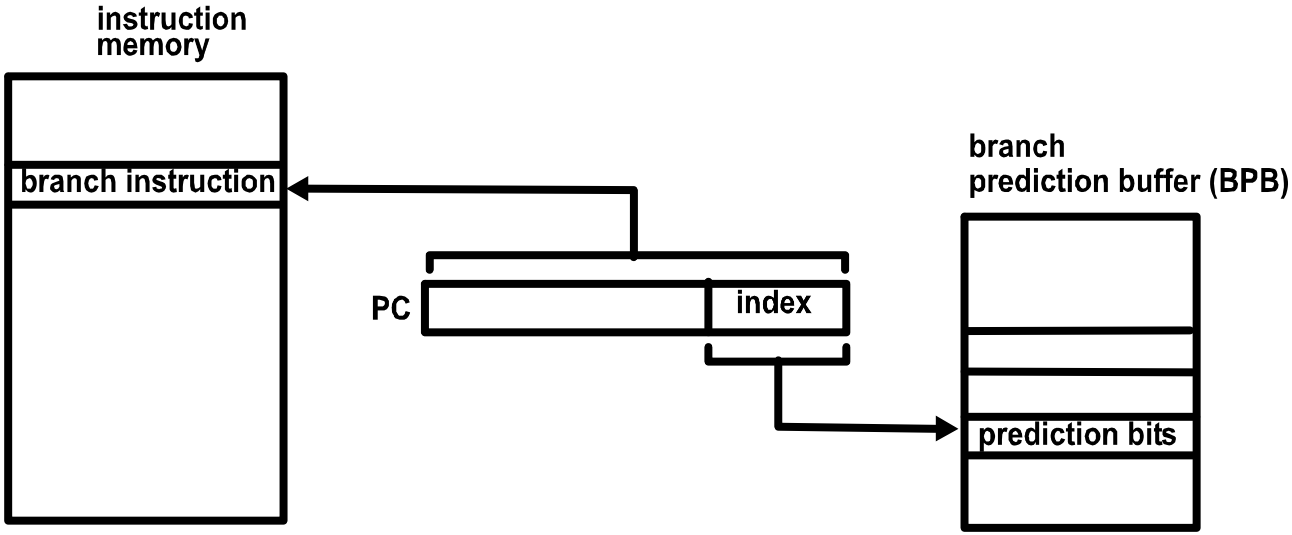
- ROLL BACK MECHANISM IN CASE PREDICTION IS WRONG

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
```

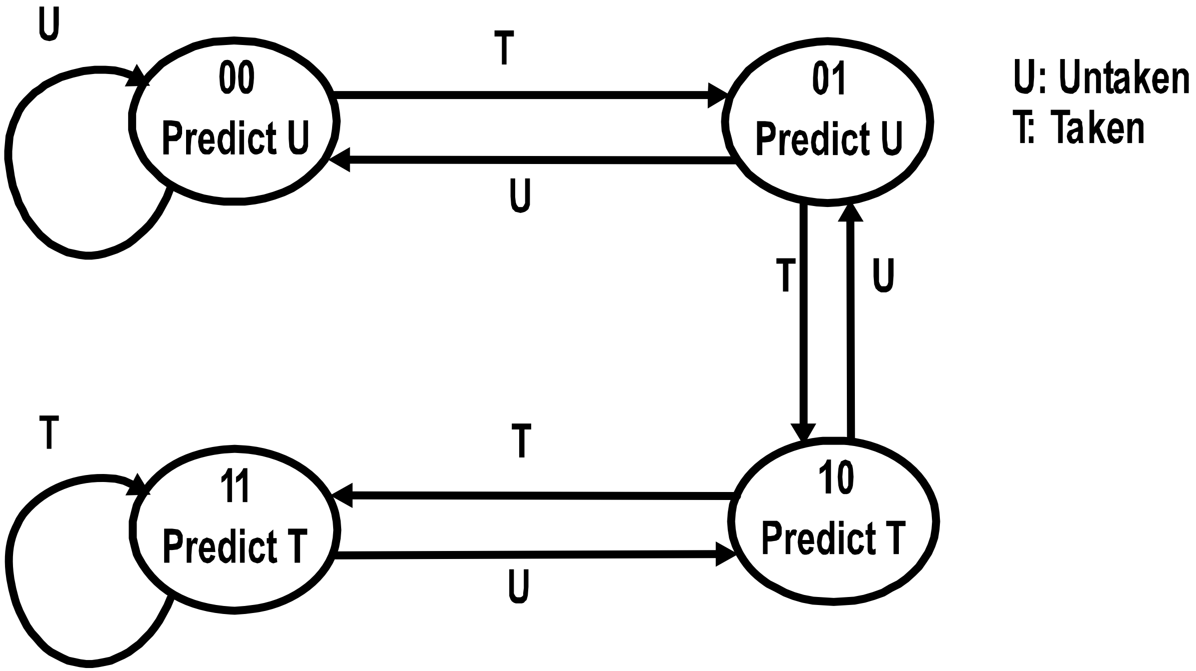
BEZ IS ALWAYS MISPREDICT TWICE FOR EVERY LOOP EXECUTION

- 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

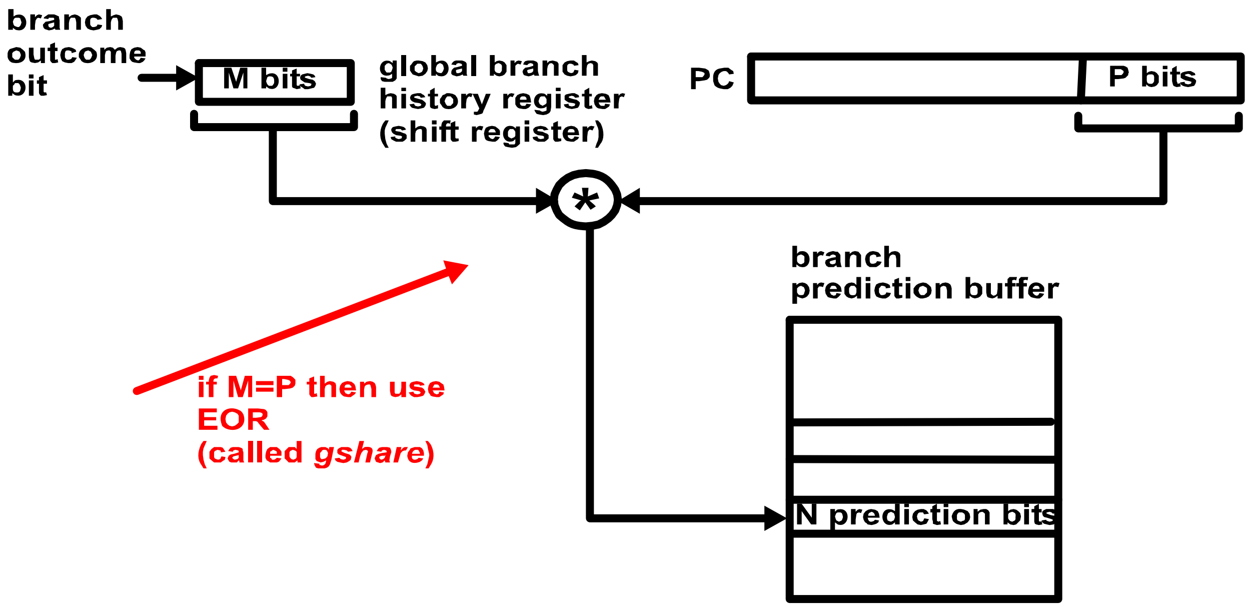
(M,N) BPB

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 X 2^M X 2^P

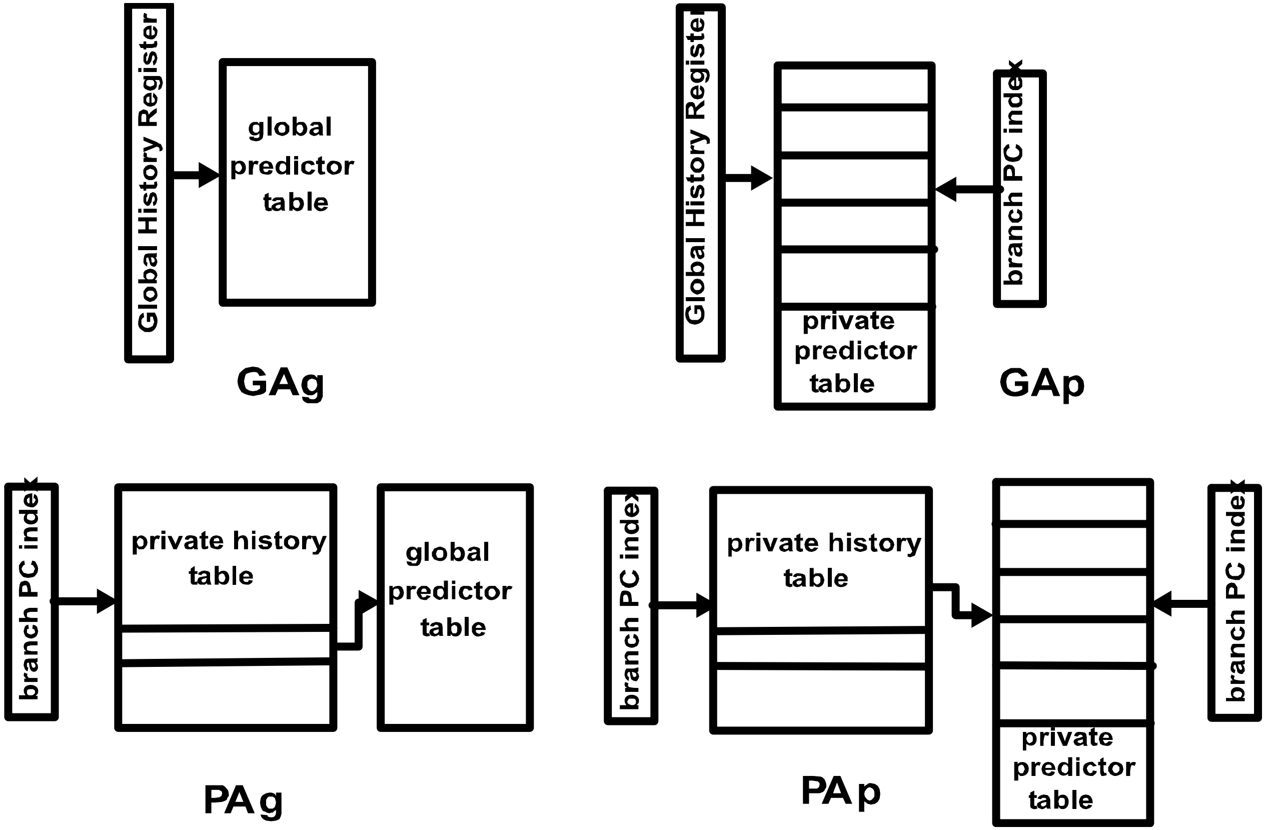


if M=P then use EOR (called *gshare*)

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



Global History Register

global predictor table

GAg

Global History Register

private predictor table

branch PC index

GAp

branch PC index

private history table

global predictor table

PAg

branch PC index

private history table

private predictor table

branch PC index

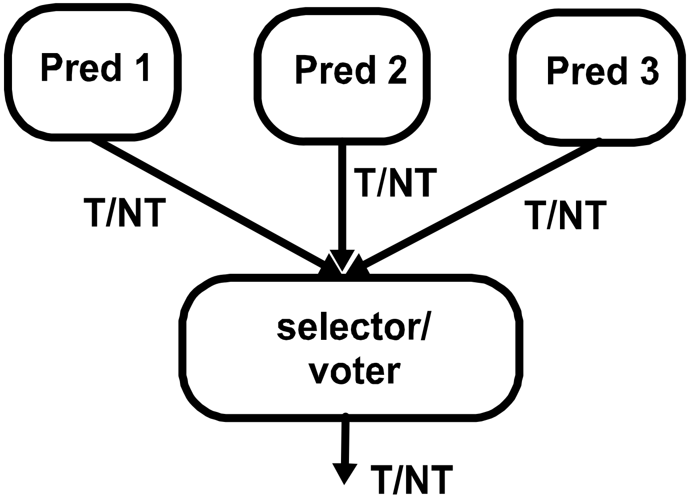
PAp

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.



Pred 1

Pred 2

Pred 3

T/NT

T/NT

T/NT

selector/
voter

T/NT

A SELECTOR KEEPS THE TRACK RECORD OF EACH PREDICTOR (GLOBALLY OR FOR EACH BRANCH). THIS CAN BE DONE WITH 1 BIT OR 2 BIT.

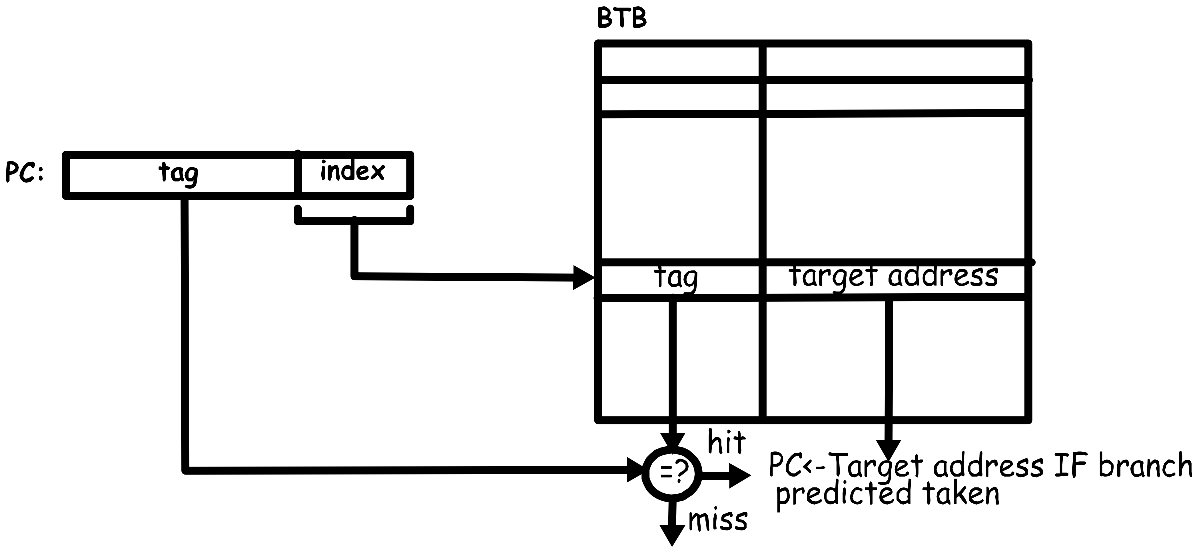
A VOTER TAKES A MAJORITY VOTE OF THE 3 PREDICTORS

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.

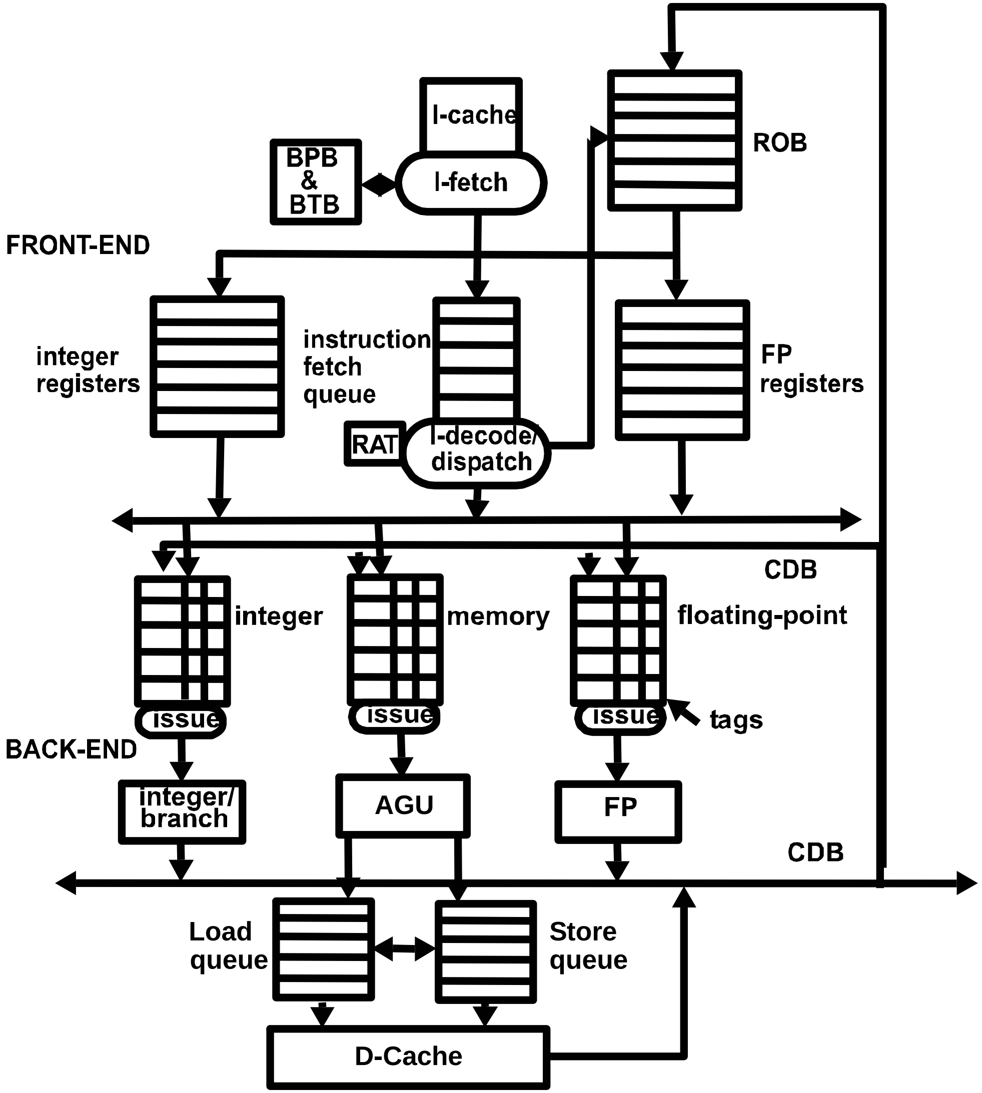
- KEY IDEAS:
  - SEPARATE THE COMPLETION OF INSTRUCTION EXECUTION AND THE COMMIT OF ITS RESULT
  - BETWEEN COMPLETION AND COMMIT RESULTS ARE SPECULATIVE
  - COMMIT RESULTS TO REGISTERS AND STORAGE IN PROCESS ORDER

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 LOADs/STOREs 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 STOREs IN FRONT OF THE LOAD WITH UNKNOWN ADDRESS (ADDRESS NOT READY)

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

EXPLICIT REGISTER RENAMING

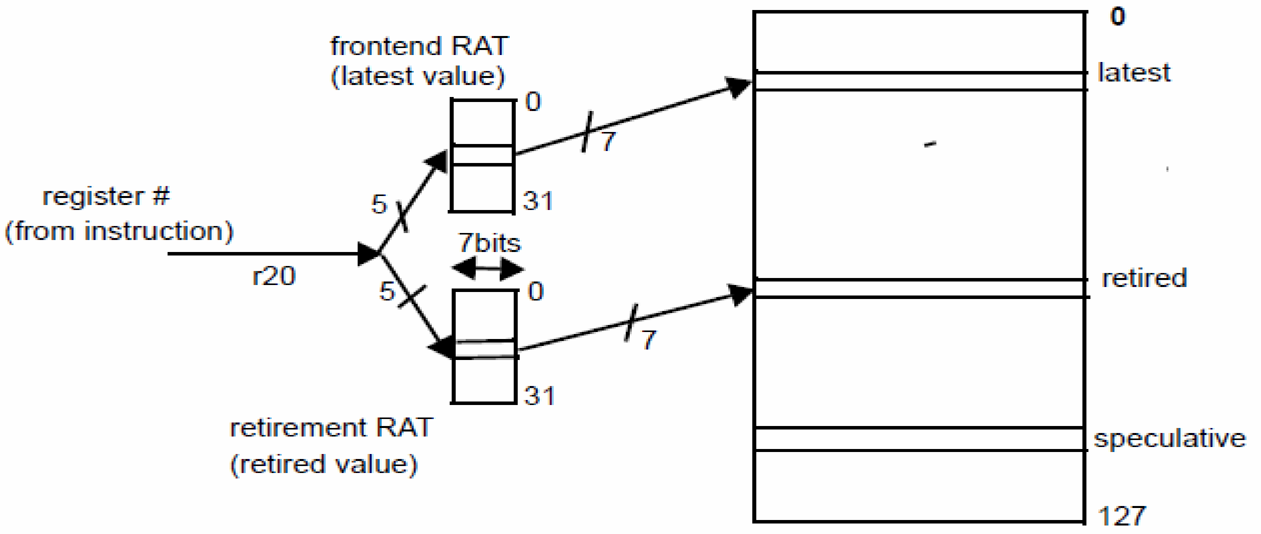
- ROB ORDERS INSTRUCTION COMMITS AND PROVIDES STORAGE FOR SPECULATIVE REGISTER VALUES UNTIL THEY COMMIT

- SPECULATIVE REGISTER VALUES MAY ALSO BE KEPT IN PHYSICAL REGISTERS.
  - THEN THE ROLE OF THE ROB IS LIMITED TO ORDERING INSTRUCTION COMMITS

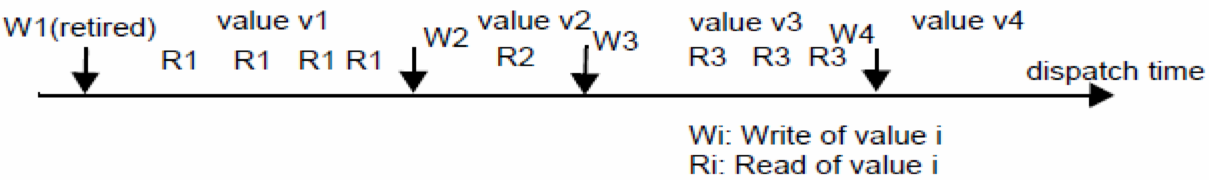
- LARGE NUMBER OF PHYSICAL REGISTERS (MORE THAN THE # OF ARCHITECTURAL REGISTERS--THE REGISTERS VISIBLE FROM THE ISA)

- 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)

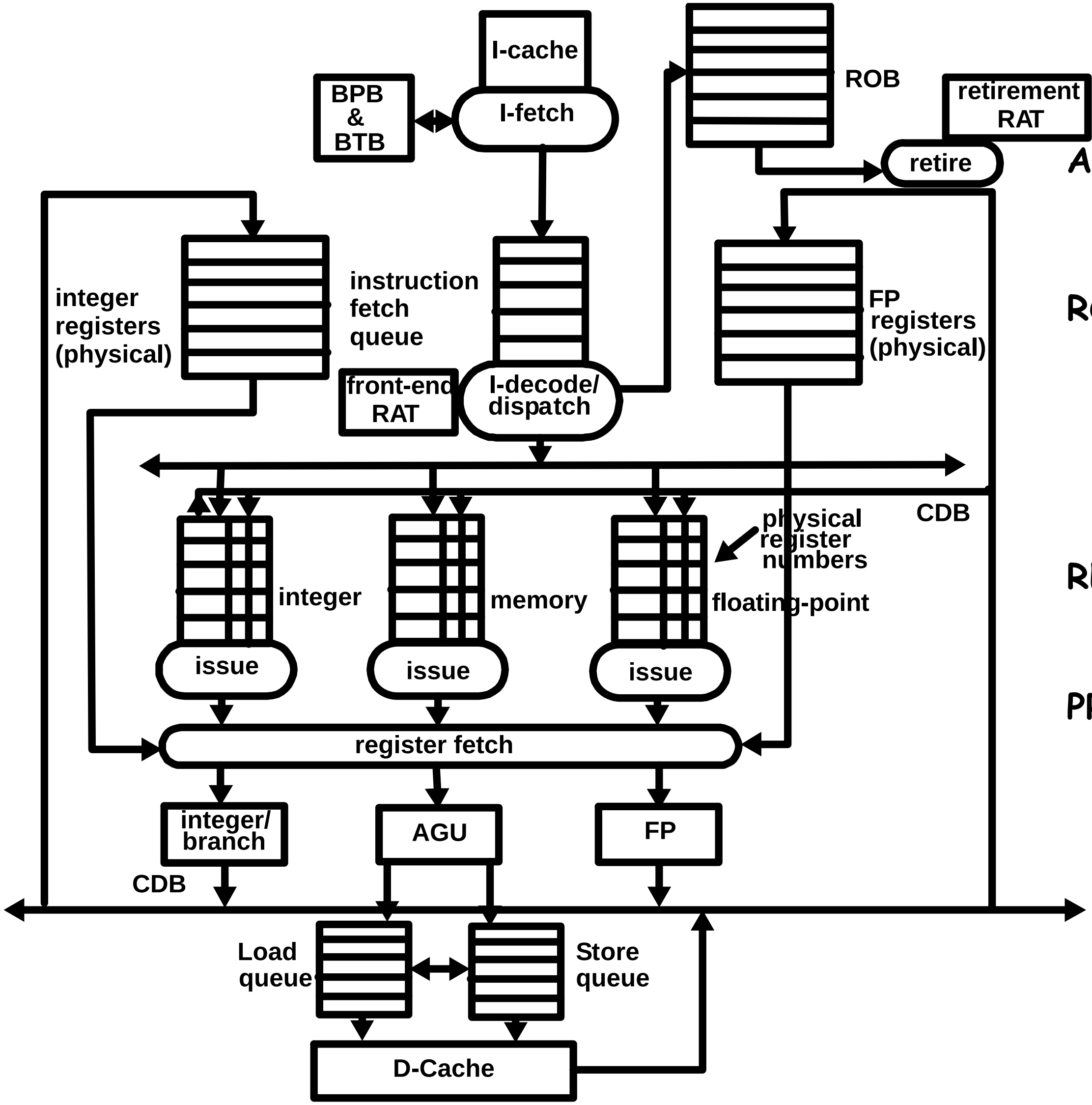


(b)

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

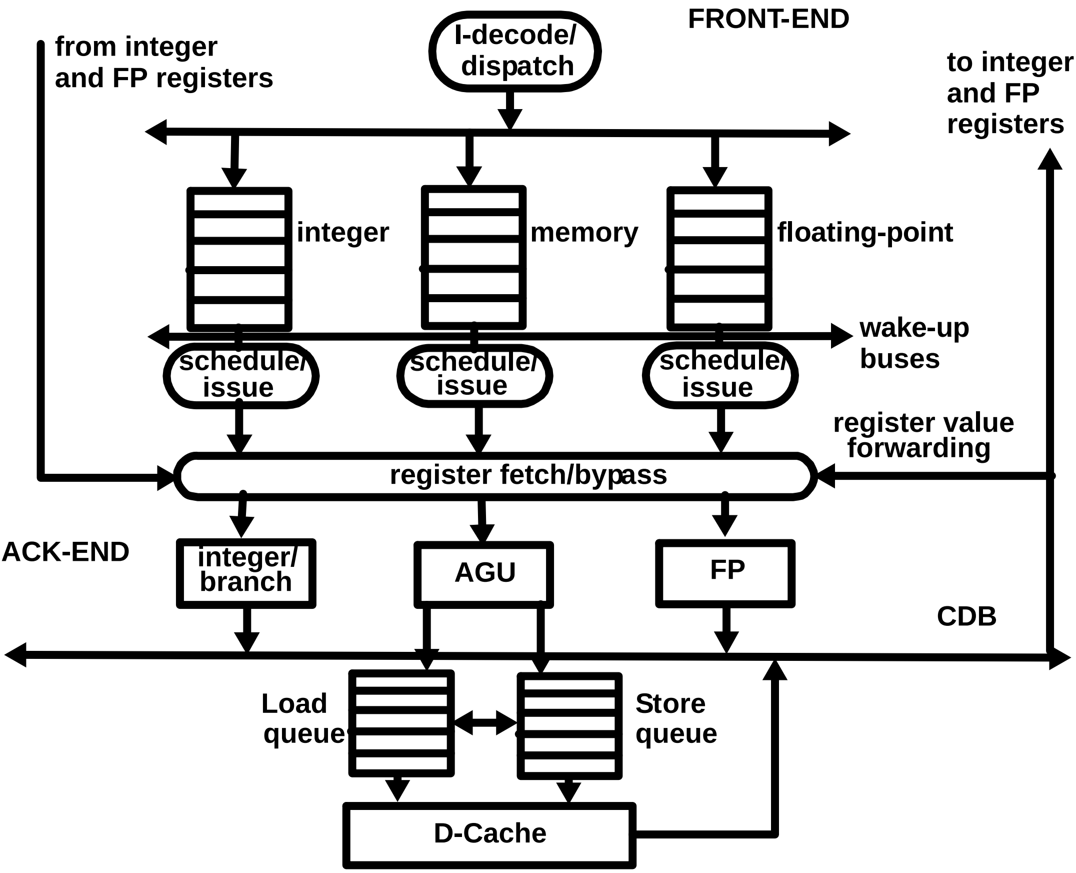
PROBLEMS:

- 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

SPECULATIVE INSTRUCTION SCHEDULING



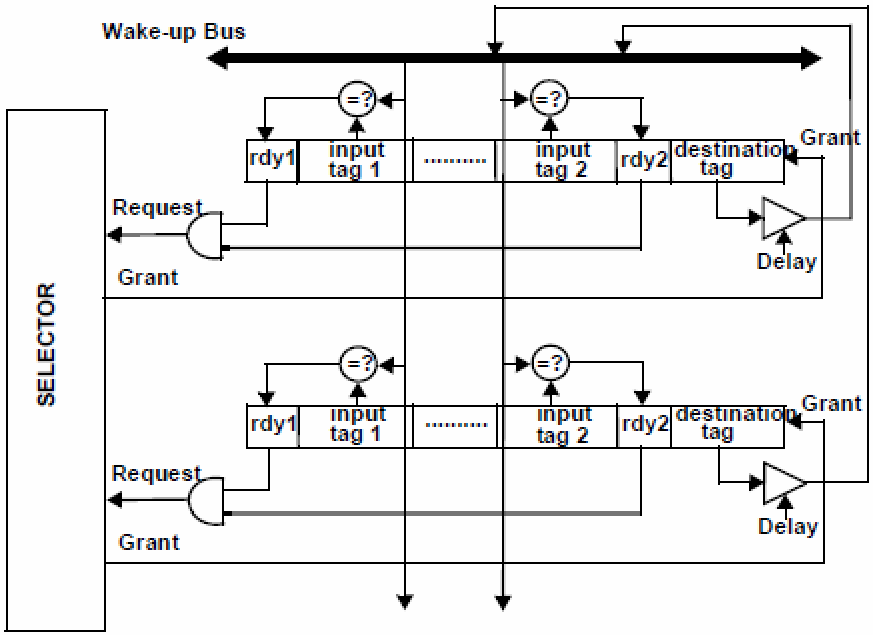
THE SCHEDULER WAKES UP OPERANDS IN ISSUE Qs

- IT SETS THE READY BITS OF WAITING OPS AFTER A NUMBE 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

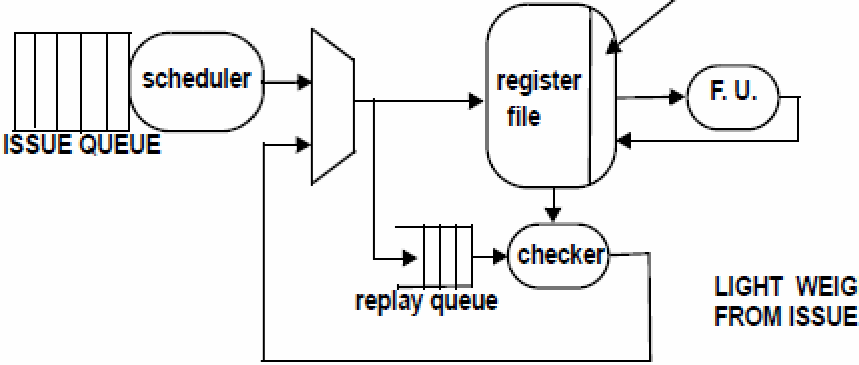
SOME INSTRUCTIONS HAVE VARIABLE LATENCIES (E.G., CACHE ACCESSES

- HIT/MISS

SPECULATIVE INSTRUCTION SCHEDULING



SCHEDULER (2 queue entries and 1 instruction issue per clock)



LIGHT WEIGHT REPLAY FROM ISSUE QUEUE

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

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.

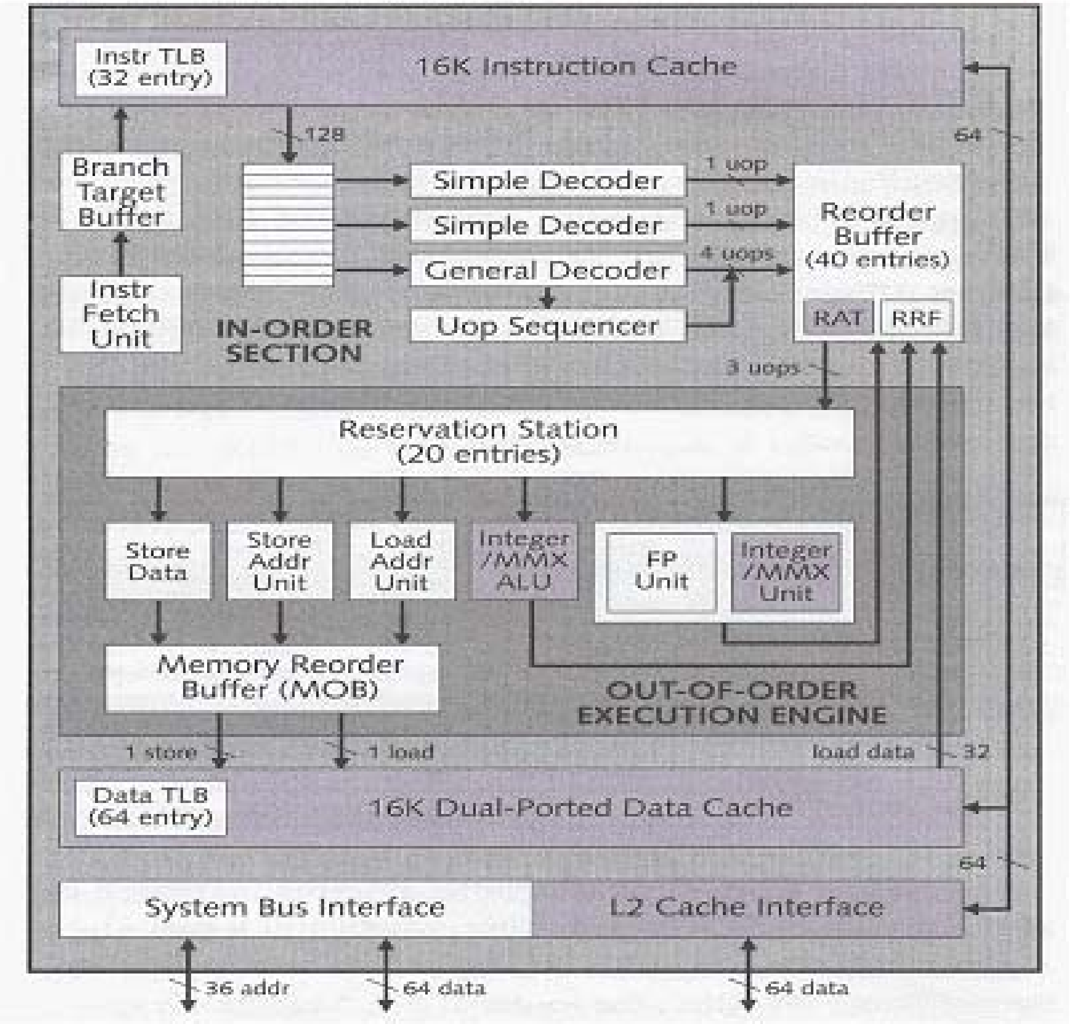
Also, 40 virtual registers and 20 reservation stations (shared)

3 stages for execution

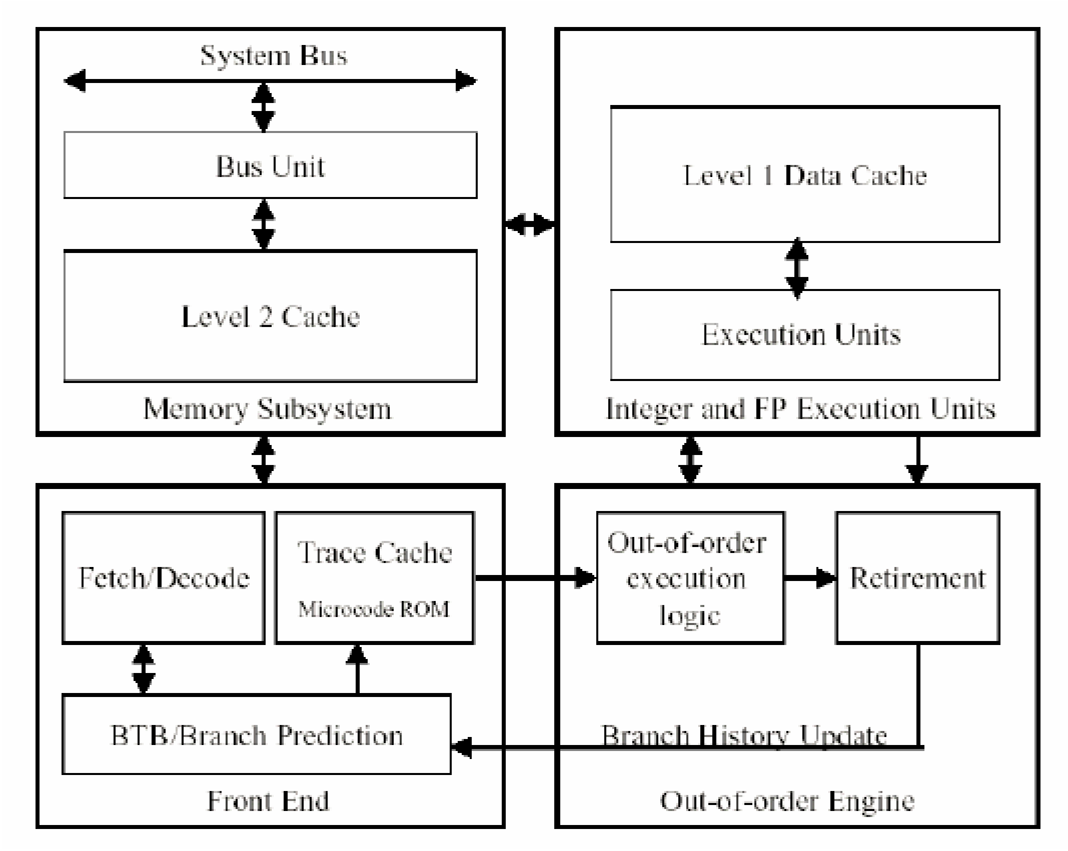
Instruction	latency	repeat
Int. ALU	0	1
Load	2	1
Int Mult	3	1
FP add	2	1
FP Mult	4	2
FP Div	31	32

Latencies and repeat intervals

INTEL PENTIUM III



INTEL PENTIUM 4



INTEL PENTIUM 4

42 million transistors

217mm^2

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 3microps 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, microps 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

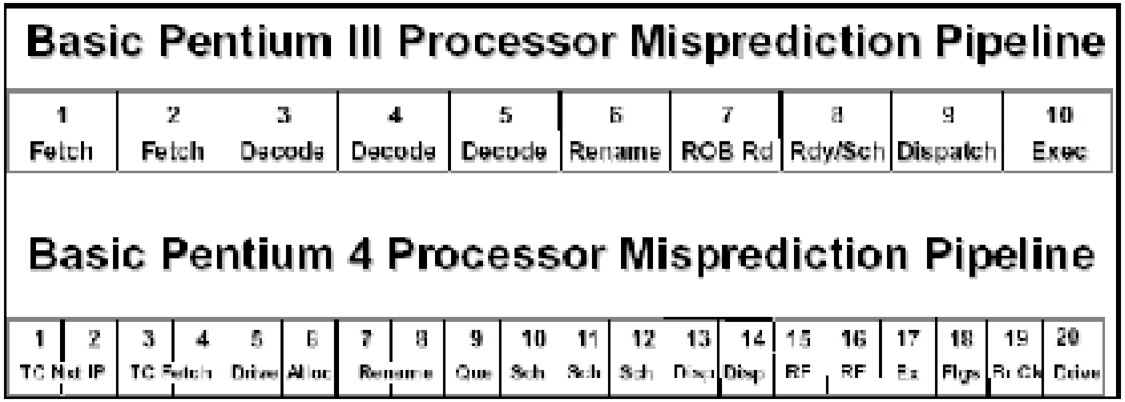
1 LD, 1 ST, 3 INT and 2 FP/MMX

Memory system

L1 8KB 4-way 128B

No I-cache

L2 Unified 256KB 8-way 128B



Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Figure 3: Misprediction Pipeline

INTEL PENTIUM 4

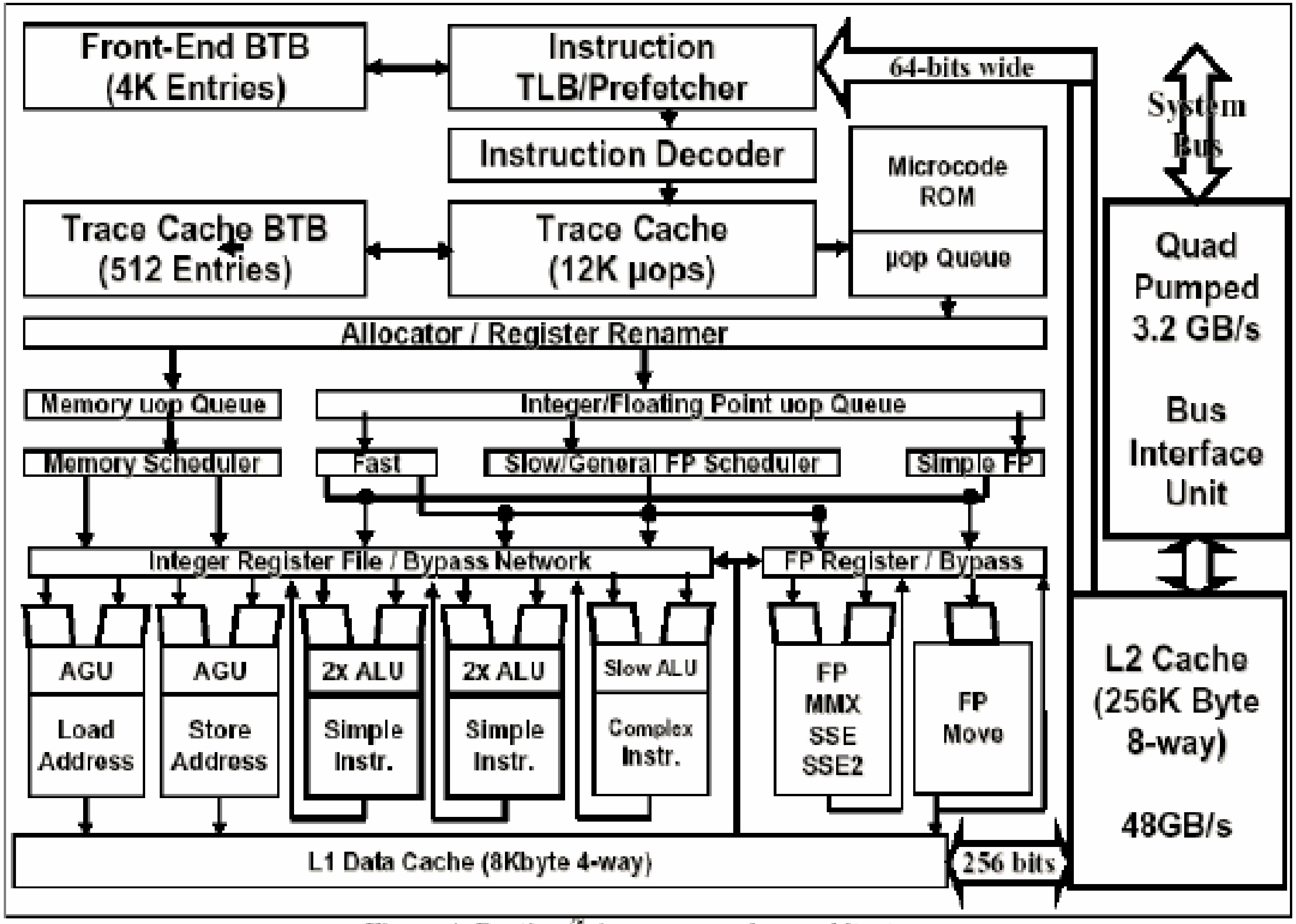


Figure 4: Pentium® 4 processor microarchitecture

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] 🡐 V2[i] + V3[i], for all i's

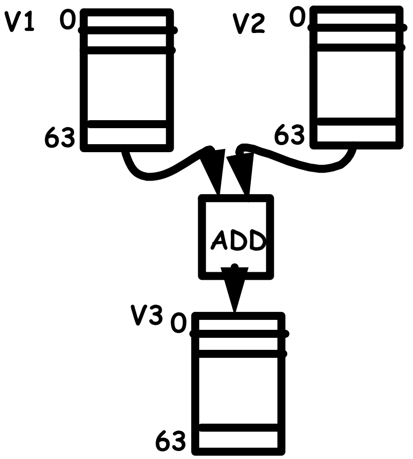
VECTOR LENGTH AND STRIDE MAY BE HELD IN SPECIAL CONTROL REGISTERS

VECTOR MACHINES EXISTED WELL BEFORE SUPERSCALAR PROCESSORS

THEY ARE STILL BUILT FOR SUPERCOMPUTERS FOR ENGINEERING/SCIENTIFIC COMPUTATION

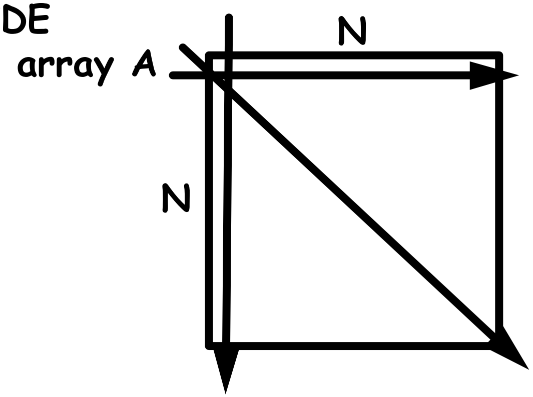
BECAUSE OF MULTIMEDIA (STREAMING) APPLICATION THEY ARE REAPPEARING IN COMMODITY MARKETS (DSP)

VECTOR PROCESSORS



Assume ADD has 10 stages
The total execution time is:
$T_{ex} = 10 + 63 = 9 + 64$
In general: $T_{ex} = T_{start} + N$

STRIDE



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

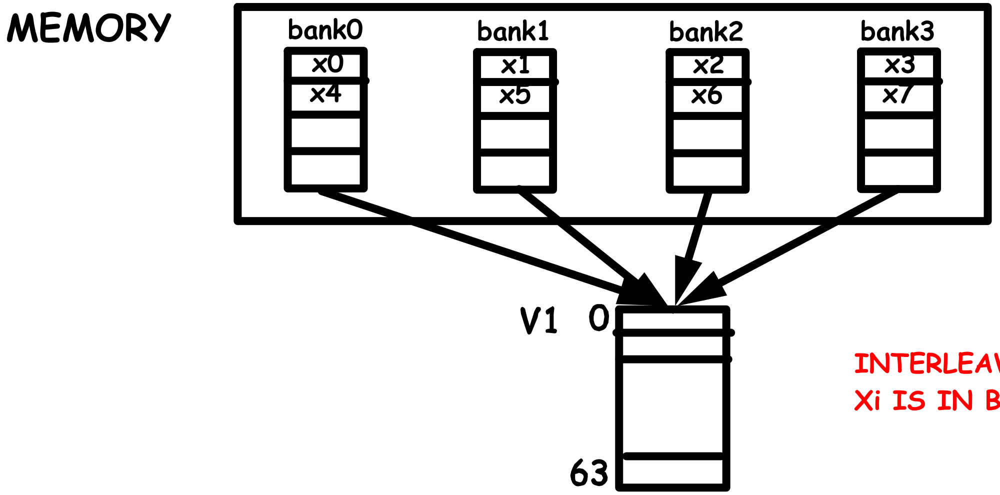
LOAD STORE UNITS CAN BE SEEN AS PIPELINES

- 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 THAT 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

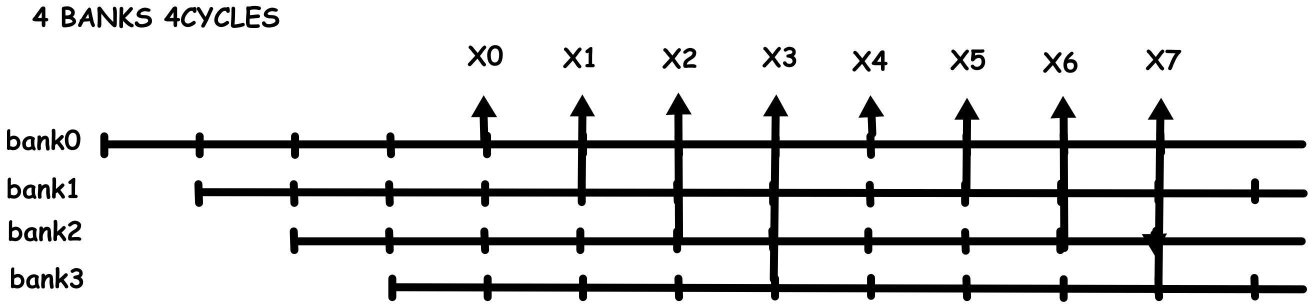
VECTOR--MEMORY ORGANIZATION

HEAVILY INTERLEAVED (HUNDREDS OF MEMORY MODULES)

SIMPLE EXAMPLE



INTERLEAVING:
Xi IS IN BANK (I MOD 4)



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

CHAINING AND PARALLEL EXECUTION

INDEPENDENT VECTOR INSTRUCTIONS CAN BE EXECUTED IN PARALLEL

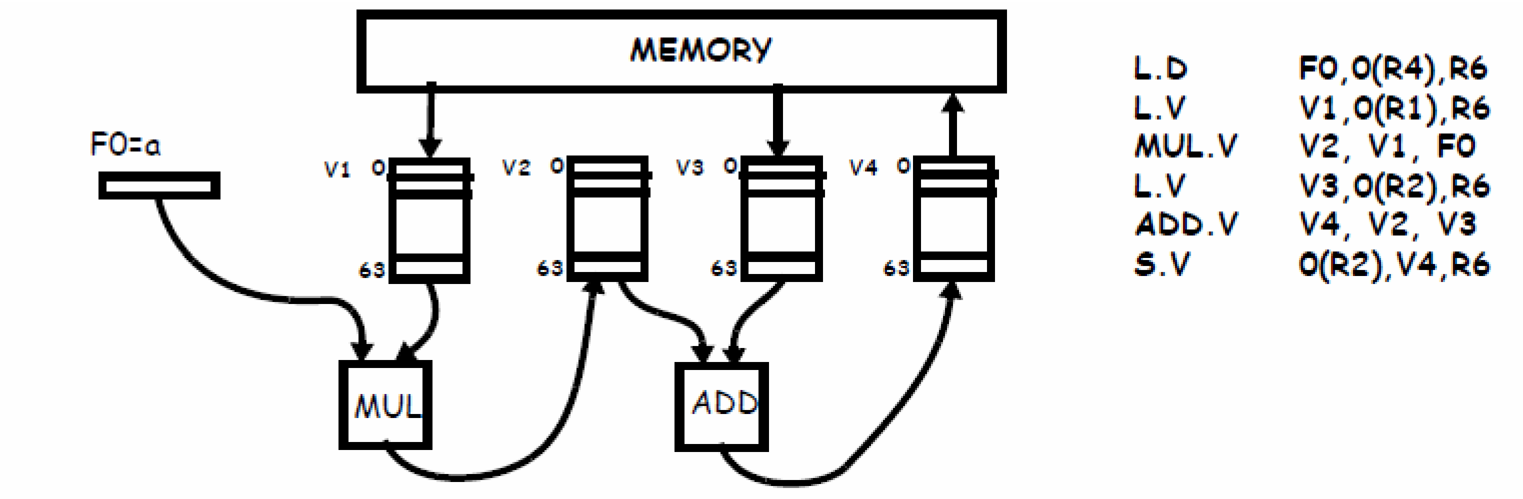
PLUS CAN BE CHAINED

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)



```
L.D      F0,0(R4),R6
L.V      V1,0(R1),R6
MUL.V    V2, V1, F0
L.V      V3,0(R2),R6
ADD.V    V4, V2, V3
S.V      0(R2),V4,R6
```

SPECIAL MECHANISMS

FOR CONDITIONAL STATMENTS

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] ==> 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))
```

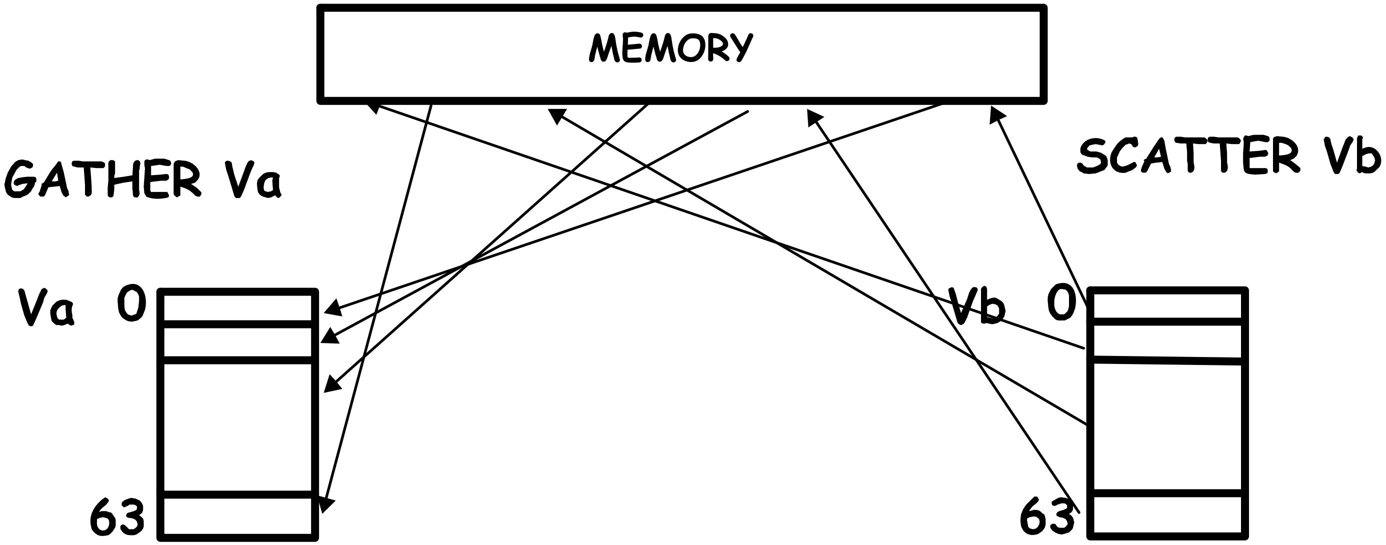
MEMORY SCATTER/GATHER

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-REGISTERTO INDEXED ADDRESSES

```
L.V Vk,0(R1),R6          /load vector K in REGISTER Vk
LI.V   Va,Vk,0(R2)                /load indexed from A(K(i)): gather
              <work on Va, put result in Vb>
SI.V   0(R2), Vk, Vb              /store indexed to A(K(i)): scatter
```


MEMORY
GATHER Va
SCATTER Vb
Va
0
63
Vb
0
63