CS152
Computer Architecture and Engineering
Lecture 14

Introduction to Advanced Pipelining

March 17, 1999

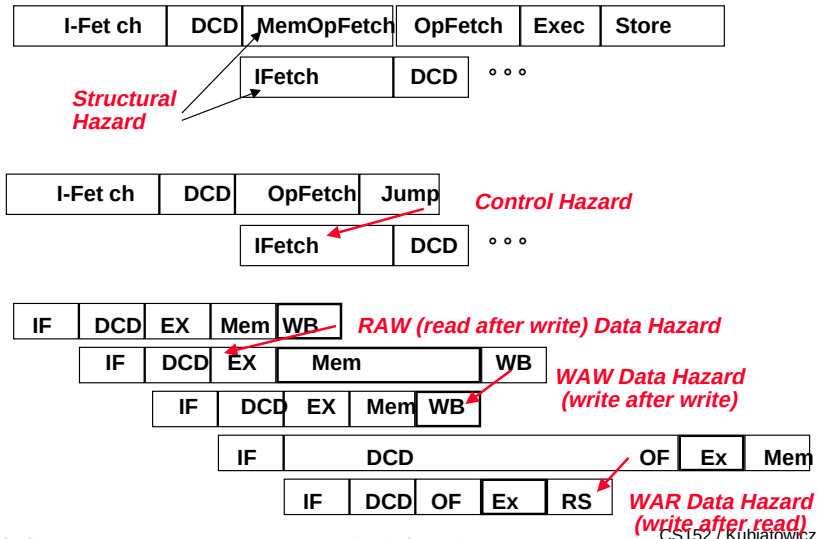
John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Review: Summary of Pipelining Basics

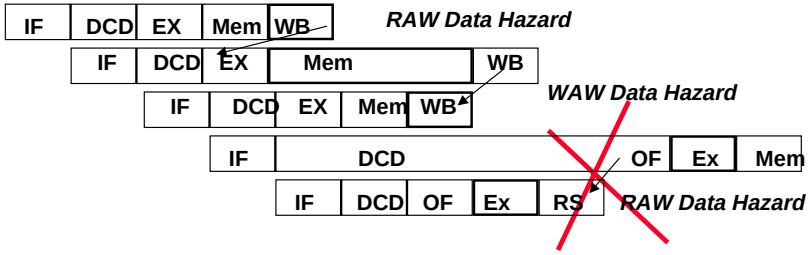
- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency

Recap: Pipeline Hazards

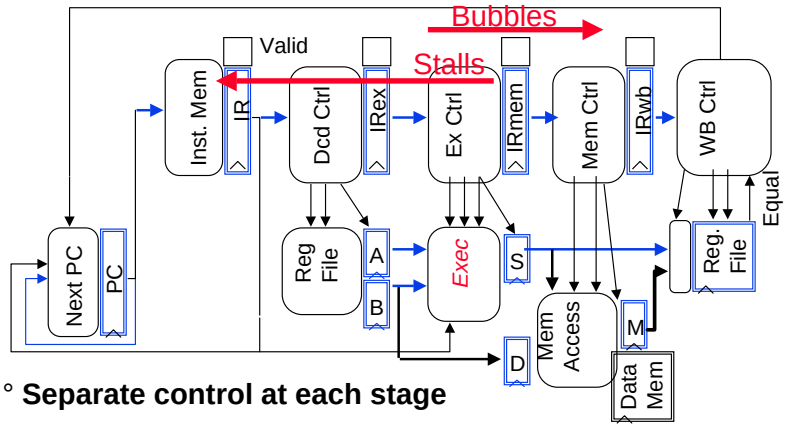


Recap: Data Hazards

- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)



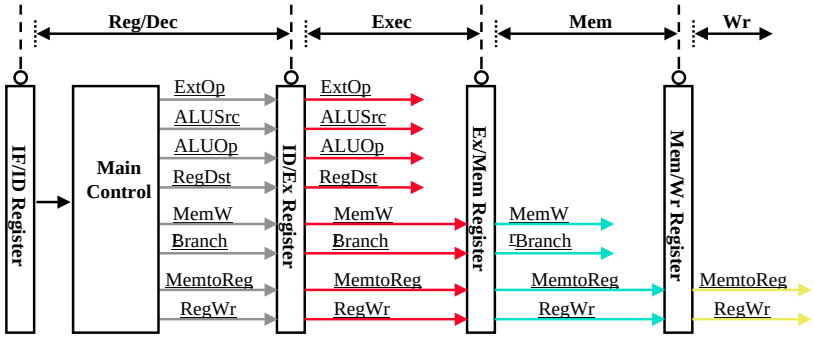
Recap: Pipelined Processor for slides



- Separate control at each stage
- Stalls propagate backwards to freeze previous stages
- Bubbles in pipeline introduced by placing "Noops" into local stage, stall previous stages.

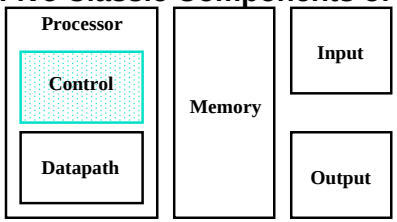
Recap: Data Stationary Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



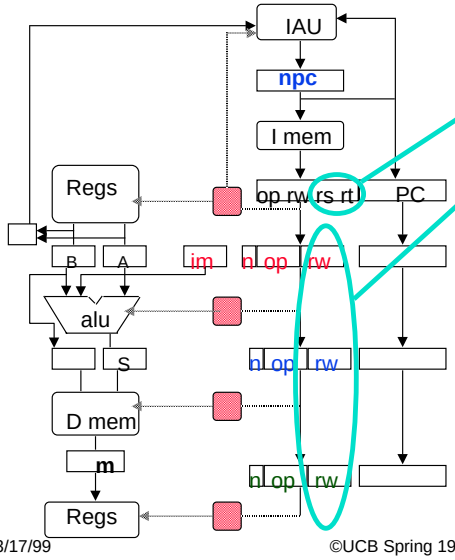
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- Today's Topics:
  - Recap last lecture
  - Review MIPS R3000 pipeline
  - Administrivia
  - Advanced Pipelining
  - SuperScalar, VLIW/EPIC

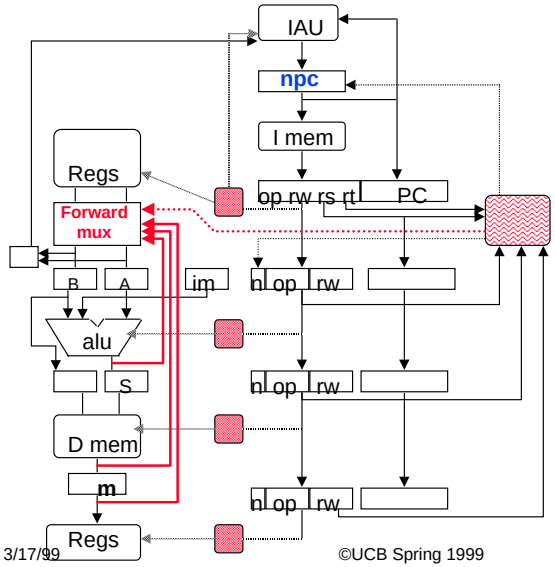
Recap: Record of Pending Writes



- Current operand registers
- Pending writes
- hazard <=

  ((rs == rw_{ex}) & $regW_{ex}$) OR
  ((rs == rw_{mem}) & $regW_{me}$) OR
  ((rs == rw_{wb}) & $regW_{wb}$) OR
  ((rt == rw_{ex}) & $regW_{ex}$) OR
  ((rt == rw_{mem}) & $regW_{me}$) OR
  ((rt == rw_{wb}) & $regW_{wb}$)

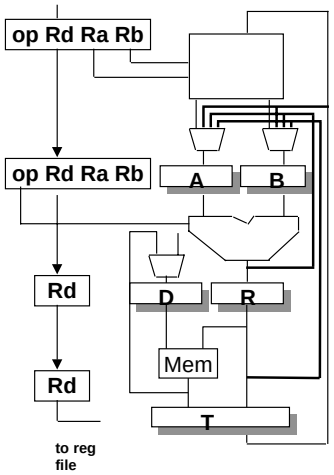
Recap: Resolve RAW by forwarding



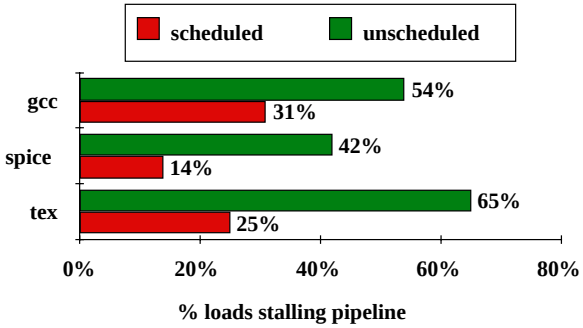
- Detect nearest **valid** write op operand register and **forward** into op latches, **bypass**ing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- **Data Forwarding** = **Data Bypassing**

What about memory operations?

- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?
- What about data dependence on loads?
  R1 <- R4 + R5
  R2 <- Mem[R2 + I]
  R3 <- R2 + R1
  ⇒ "Delayed Loads"
- Can recognize this in decode stage and introduce bubble while stalling fetch stage (hint for lab 5!)
- Tricky situation:
  R1 <- Mem[R2 + I]
  Mem[R3+34] <- R1
  Handle with bypass in memory stage!



Compiler Avoiding Load Stalls:



What about Interrupts, Traps, Faults?

- **External Interrupts:**
  - Allow pipeline to drain,
  - Load PC with interrupt address
- **Faults (within instruction, restartable)**
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state
- **Recall: Precise Exceptions ⇒ State of the machine is preserved as if program executed up to the offending instruction**
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations

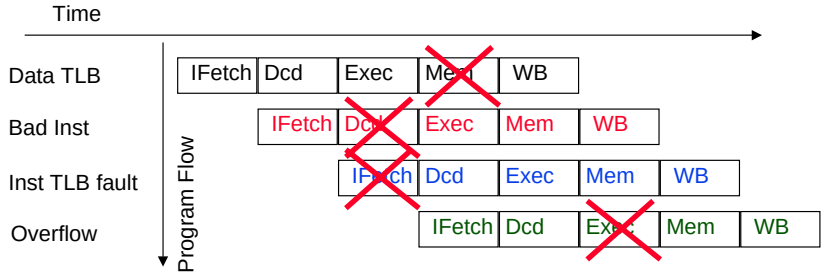
Exception Problem

- Exceptions/Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

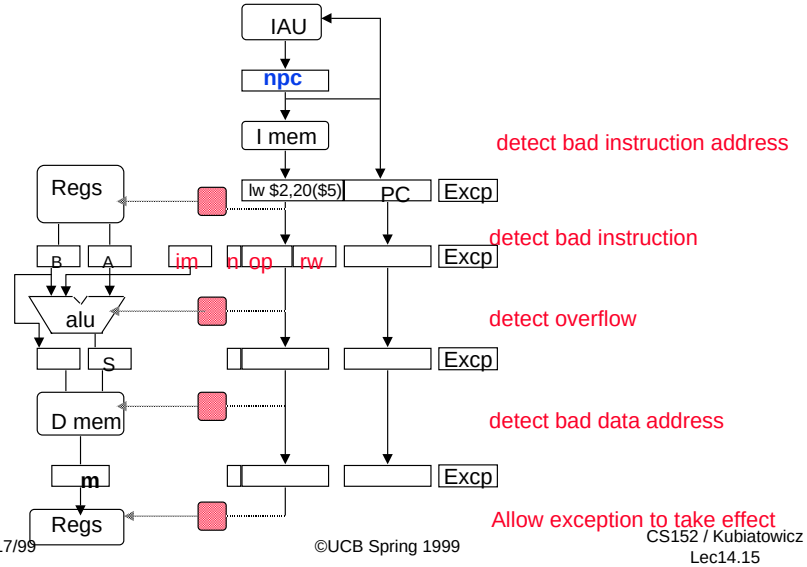
- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction 2: interrupt ASAP, restart everything incomplete

Another look at the exception problem

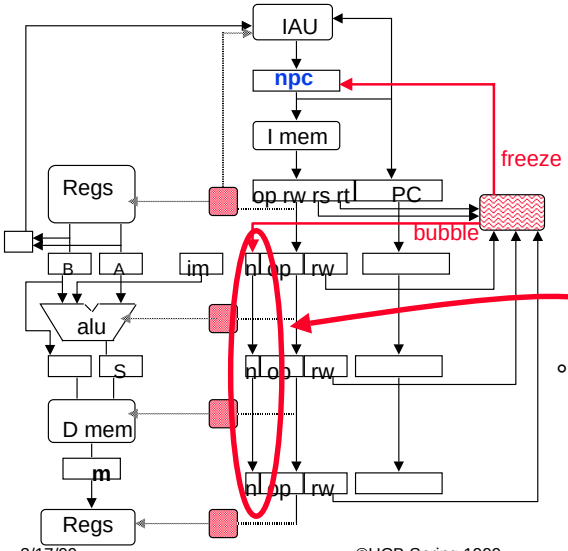


- Use pipeline to sort this out!
  - Pass exception status along with instruction.
  - Keep track of PCs for every instruction in pipeline.
  - Don't act on exception until it reache WB stage
- Handle interrupts through "faulting noop" in IF stage
- When instruction reaches WB stage:
  - Save PC ⇒ EPC, Interrupt vector addr ⇒ PC
  - Turn all instructions in earlier stages into noops!

Exception Handling



Resolution: Freeze above & Bubble Below



- Flush accomplished by setting "invalid" bit in pipeline

Administrivia

° **Policy on Homework Quizes:**
- Assuming that you have DONE homework and thought about it
- Testing your understanding.
- Will throw out lowest score at end of term.
- Don't do homework on weekend before! This way you can ask TAs about problems that you don't understand.

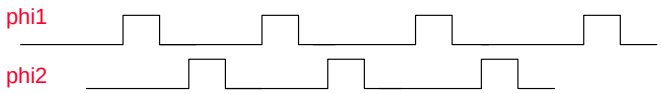
° **Mail Lab 5 breakdowns to your TAs by tonight!**

° **Get started on Lab 5: Pipelining is difficult to get right! Be sure that we will test "gotcha" cases in our mystery programs…**

° **Will hand out paper on testing (if I can find it) next class (Doug Clark on VAX).**

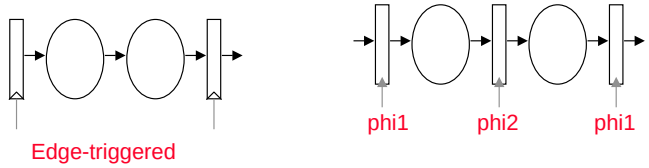
° **Next week: advanced pipelining**
- Out-of-order execution/register renaming
- Reorder buffers

FYI: MIPS R3000 clocking discipline

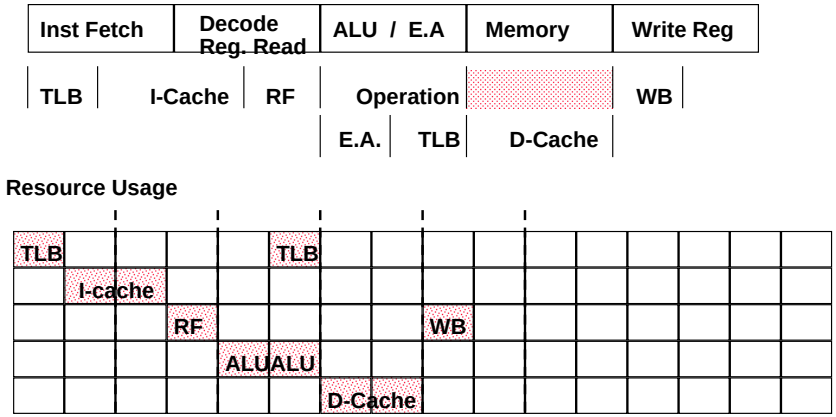


° **2-phase non-overlapping clocks**

° **Pipeline stage is two (level sensitive) latches**

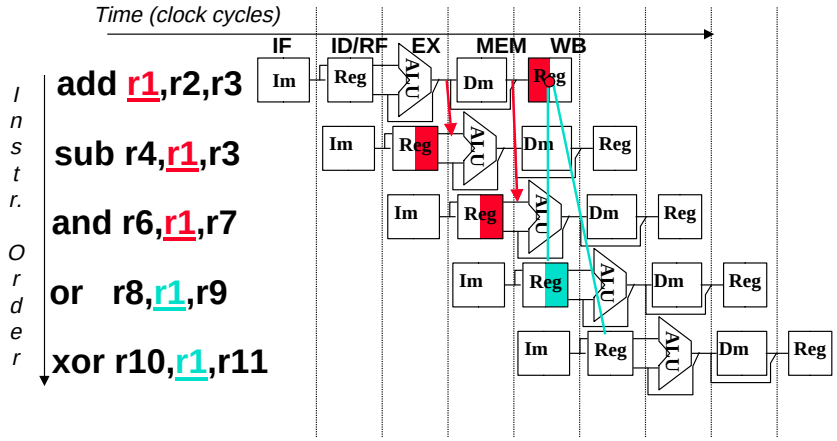


MIPS R3000 Instruction Pipeline



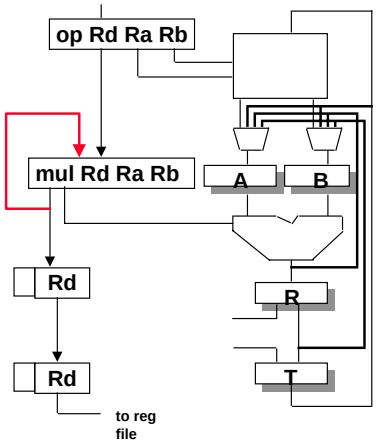
Write in phase 1, read in phase 2 => eliminates bypass from WB

Recall: Data Hazard on r1



With MIPS R3000 pipeline, no need to forward from WB stage

MIPS R3000 Multicycle Operations



Use control word of local stage to step through multicycle operation

Stall all stages above multicycle operation in the pipeline

Drain (bubble) stages below it

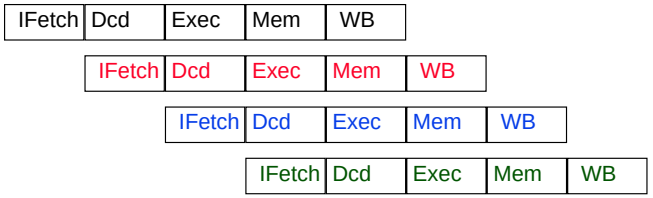
Alternatively, launch multiply/divide to autonomous unit, only stall pipe if attempt to get result before ready

- This means stall mflo/mfhi in decode stage if multiply/divide still executing
- Extra credit in Lab 5 does this

Ex: Multiply, Divide, Cache Miss

Is CPI = 1 for our pipeline?

° Remember that CPI is an "Average # cycles/inst



° CPI here is 1, since the average throughput is 1 instruction every cycle.

° What if there are stalls or multi-cycle execution?

° Usually CPI > 1. How close can we get to 1??

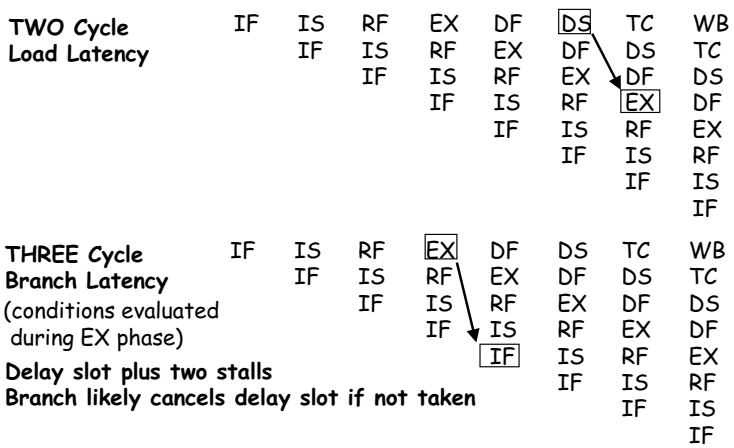
Case Study: MIPS R4000 (200 MHz)

° 8 Stage Pipeline:

- IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
- IS–second half of access to instruction cache.
- RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
- EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
- DF–data fetch, first half of access to data cache.
- DS–second half of access to data cache.
- TC–tag check, determine whether the data cache access hit.
- WB–write back for loads and register-register operations.

° 8 Stages: What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000



MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

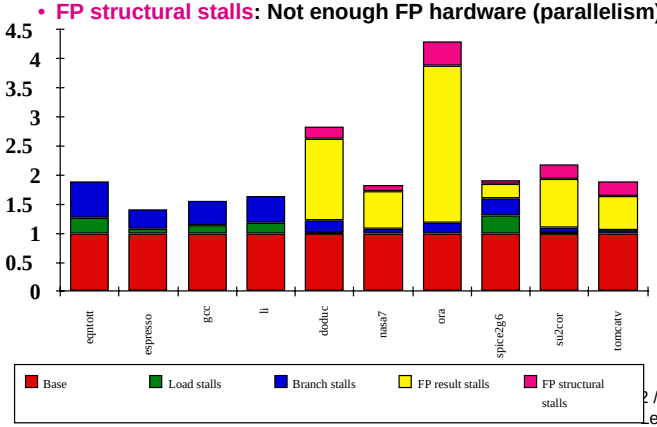
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*…*
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R,	D+R,	D+A, D+R, A, R
Square root	U	E	$(A+R)^{108}$		…	A	R		
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*	*A*	*Mantissa ADD stage*
N	*Second stage of multiplier*	*D*	*Divide pipeline stage*
R	*Rounding stage*	*E*	*Exception test stage*
S	*Operand shift stage*		
U	*Unpack FP numbers*		

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



Advanced Pipelining and Instruction Level Parallelism (ILP)

- ILP: Overlap execution of unrelated instructions
- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Beyond single block to get more instruction level parallelism
- Loop level parallelism one opportunity
  - First SW, then HW approaches
- DLX Floating Point as example
  - Measurements suggests R4000 performance FP execution has room for improvement

FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar from F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: LD    F0,0(R1)   ;F0=vector element
2       stall
3       ADDD  F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       SD    0(R1),F4   ;store result
7       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
8       BNEZ  R1,Loop    ;branch R1!=zero
9       stall            ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop    ;delayed branch
6       SD    8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop:LD     F0,0(R1)        1 cycle stall
2       ADDD   F4,F0,F2        2 cycles stall
3       SD     0(R1),F4        ;drop SUBI & BNEZ
4       LD     F6,-8(R1)
5       ADDD   F8,F6,F2
6       SD     -8(R1),F8       ;drop SUBI & BNEZ
7       LD     F10,-16(R1)
8       ADDD   F12,F10,F2
9       SD     -16(R1),F12     ;drop SUBI & BNEZ
10      LD     F14,-24(R1)
11      ADDD   F16,F14,F2
12      SD     -24(R1),F16
13      SUBI   R1,R1,#32       ;alter to 4*8
14      BNEZ   R1,LOOP
15      NOP
```

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16     ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

When safe to move instructions?

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two main variations: Superscalar and VLIW
- Superscalar: varying no. instructions/cycle (1 to 6)
  - Parallelism and dependencies determined/resolved by HW
  - IBM PowerPC 604, Sun UltraSparc, DEC Alpha 21164, HP 7100
- Very Long Instruction Words (VLIW): fixed number of instructions (16) parallelism determined by compiler
  - Pipeline is exposed; compiler must schedule delays to get right result
- Explicit Parallel Instruction Computer (EPIC)/ Intel
  - 128 bit packets containing 3 instructions (can execute sequentially)
  - Can link 128 bit packets together to allow more parallelism
  - Compiler determines parallelism, HW checks dependencies and fowards/stalls

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*PipeStages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

Unrolled Loop that Minimizes Stalls for Scalar

```
1 Loop: LD     F0,0(R1)
2       LD     F6,-8(R1)
3       LD     F10,-16(R1)
4       LD     F14,-24(R1)
5       ADDD   F4,F0,F2
6       ADDD   F8,F6,F2
7       ADDD   F12,F10,F2
8       ADDD   F16,F14,F2
9       SD     0(R1),F4
10      SD     -8(R1),F8
11      SD     -16(R1),F12
12      SUBI   R1,R1,#32
13      BNEZ   R1,LOOP
14      SD     8(R1),F16     ; 8-32 = -24
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

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

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
- VLIW: tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Loop Unrolling in VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op/ branch	Clock
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

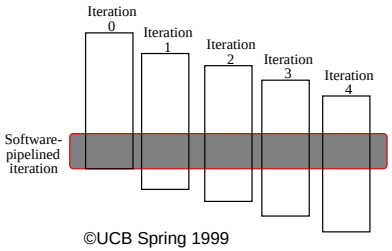
Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration

Need more registers in VLIW(EPIC => 128int + 128FP)

Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (- Tomasulo in SW)



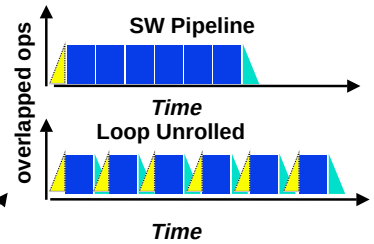
Software Pipelining Example

Before: Unrolled 3 times

```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

After: Software Pipelined

```
1   SD    0(R1),F4 ; Stores M[i]
2   ADDD  F4,F0,F2 ; Adds to M[i-1]
3   LD    F0,-16(R1); Loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LOOP
```



- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

Software Pipelining with Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,-48(R1)	ST 0(R1),F4	ADDD F4,F0,F2			1
LD F6,-56(R1)	ST -8(R1),F8	ADDD F8,F6,F2		SUBI R1,R1,#24	2
LD F10,-40(R1)	ST 8(R1),F12	ADDD F12,F10,F2		BNEZ R1,LOOP	3

° Software pipelined across 9 iterations of original loop
- In each iteration of above loop, we:
  - Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
  - Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
  - Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)

° 9 results in 9 cycles, or 1 clock per iteration

° Average: 3.3 ops per clock, 66% efficiency

Note: Need less registers for software pipelining

(only using 7 registers here, was using 15)

Trace Scheduling

° Parallelism across IF branches vs. LOOP branches

° Two steps:
- *Trace Selection*
  - Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
- *Trace Compaction*
  - Squeeze trace into few VLIW instructions
  - Need bookkeeping code in case prediction is wrong

° Compiler undoes bad guess (discards values in registers)

° Subtle compiler bugs mean wrong answer vs. pooer performance; no hardware interlocks

HW Schemes: Instruction Parallelism

° Why in HW at run time?
- Works when can't know real dependence at compile time
- Compiler simpler
- Code for one machine runs well on another

° 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 => out-of-order completion
- ID stage checked both for structural

HW Schemes: Instruction Parallelism

- Out-of-order execution divides ID stage:
  1. Issue—decode instructions, check for structural hazards
  2. Read operands—wait until no data hazards, then read operands
- Scoreboards allow instruction to execute whenever 1 & 2 hold, not waiting for prior instructions
- CDC 6600: In order issue, out of order execution, out of order commit (also called completion)

Scoreboard Implications

- Out-of-order completion => WAR, WAW hazards?
- Solutions for WAR
  - Queue both the operation and copies of its operands
  - Read registers only during Read Operands stage
- For WAW, must detect hazard: stall until other completes
- Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units
- Scoreboard keeps track of dependencies, state or operations
- Scoreboard replaces ID, EX, WB with 4 stages

Performance of Dynamic SS

Iteration no.	*Instructions*	*Issues*	*Executes*	*Writes result*
		clock-cycle number		
1	LD F0,0(R1)	1	2	4
1	ADDD F4,F0,F2	1	5	8
1	SD 0(R1),F4	2	9	
1	SUBI R1,R1,#8	3	4	5
1	BNEZ R1,LOOP	4	5	
2	LD F0,0(R1)	5	6	8
2	ADDD F4,F0,F2	5	9	12
2	SD 0(R1),F4	6	13	
2	SUBI R1,R1,#8	7	8	9
2	BNEZ R1,LOOP	8	9	

- 4 clocks per iteration

Branches, Decrements still take 1 clock cycle

Prediction: Branches, Dependencies, Data
New era in computing?

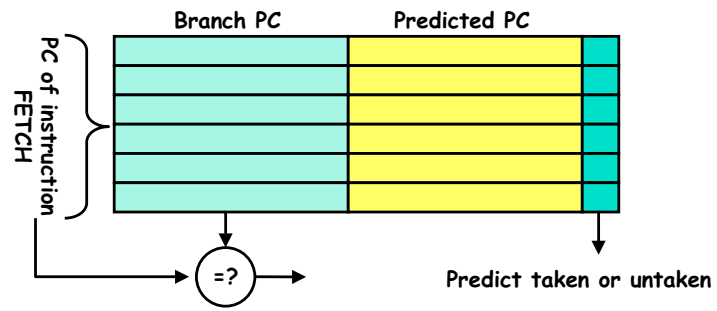
- Prediction has become essential to getting good performance from scalar instruction streams.
- We will discuss predicting branches, data dependencies, actual data, and results of groups of instructions:
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty close with control hazards already…
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
- Prediction $\Rightarrow$ Compressible information streams?

Dynamic Branch Prediction

- Is dynamic branch prediction better than static branch prediction?
  - Seems to be. Still some debate to this effect
  - Josh Fisher had good paper on "Predicting Conditional Branch Directions from Previous Runs of a Program." ASPLOS '92. In general, good results if allowed to run program for lots of data sets.
    - How would this information be stored for later use?
    - Still some difference between best possible static prediction (using a run to predict itself) and weighted average over many different data sets
  - Paper by Young et all, "A Comparative Analysis of Schemes for Correlated Branch Prediction" notices that there are a small number of important branches in programs which have dynamic behavior.

Need Address at Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address (Figure 4.22, p. 273)



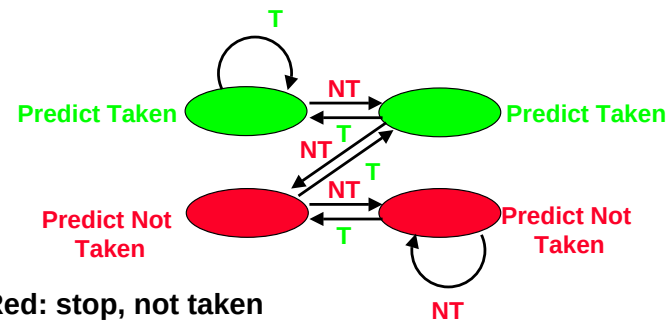
- Return instruction addresses predicted with stack

Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iteratios before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:* (Figure 4.13, p. 264)



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4096 about as good as infinite table (in Alpha 211164)

Correlating Branches

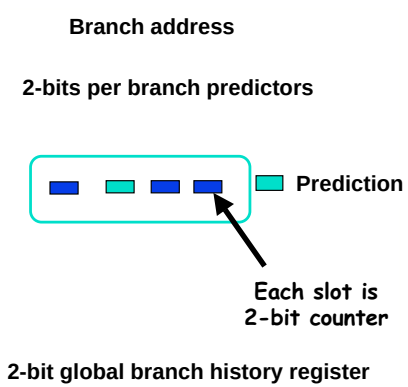
- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Two possibilities; Current branch depends on:
  - Last m most recently executed branches anywhere in program Produces a "GA" (for "global address") in the Yeh and Patt classification (e.g. GAg)
  - Last m most recent outcomes of same branch. Produces a "PA" (for "per address") in same classification (e.g. PAg)
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table entry
  - A single history table shared by all branches (appends a "g" at end), indexed by history value.
  - Address is used along with history to select table entry (appends a "p" at end of classification)
  - If only portion of address used, often appends an "s" to indicate "set-indexed" tables (I.e. GAs)

Correlating Branches

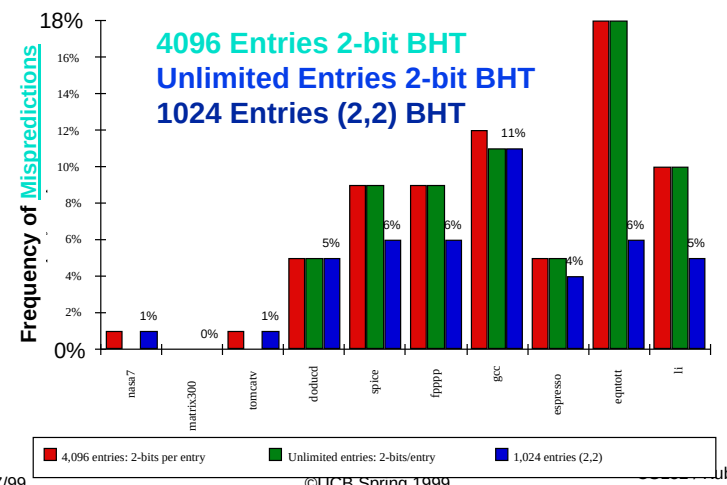
- For instance, consider global history, set-indexed BHT. That gives us a GAs history table.

(2,2) GAs predictor

- First 2 means that we keep two bits of history
- Second means that we have 2 bit counters in each slot.
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction
- Note that the original two-bit counter solution would be a (0,2) GAs predictor
- Note also that aliasing is possible here...



Accuracy of Different Schemes



Dynamic Branch Prediction Summary

- Branch History Table: 2 bits for loop accuracy
- Branch Target Buffer: include branch address & prediction

HW support for More ILP

- Avoid branch prediction by turning branches into conditionally executed instructions:

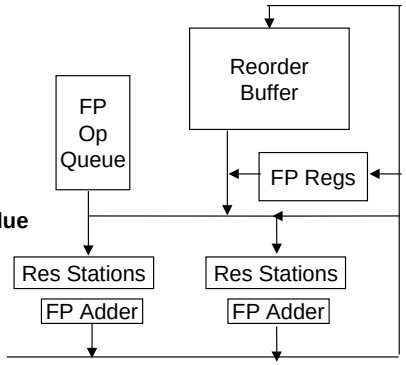
  if (x) then A = B op C else NOP
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
  - EPIC: 64 1-bit condition fields selected so conditional execution
- Drawbacks to conditional instructions
  - Still takes a clock even if "annulled"
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline

HW support for More ILP

- *Speculation*: allow an instruction *without* any consequences (including exceptions) to execute if branch is not actually taken ("HW undo")
- Often try to combine with dynamic scheduling
- Separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write results (*instruction commit*)
  - execute out-of-order but commit in order

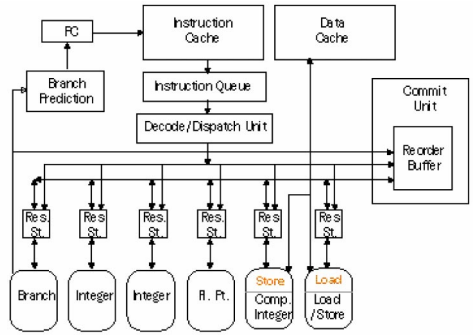
HW support for More ILP

- Need HW buffer for results of uncommitted instructions: *reorder buffer*
  - Reorder buffer can be operand source
  - Once operand commits, result is found in register
  - 3 fields: instr. type, destination, value
  - Use reorder buffer number instead of reservation station
  - Instructionsd instructions on mispredicted branches or on exceptions



Dynamic Scheduling in PowerPC 604 and Pentium Pro

° Both In-order Issue, Out-of-order execution, In-order Commit



PPro central reservation station for any functional units with one bus shared by a branch and an integer unit

Dynamic Scheduling in PowerPC 604 and Pentium Pro

Parameter	PPC	PPro
Max. instructions issued/clock	4	3
Max. instr. complete exec./clock	6	5
Max. instr. commited/clock	6	3
Instructions in reorder buffer	16	40
Number of rename buffers	12 Int/8 FP	40
Number of reservations stations	12	20
No. integer functional units (FUs)	2	2
No. floating point FUs	1	1
No. branch FUs	1	1
No. complex integer FUs	1	0
No. memory FUs	1	1 load +1 store

Dynamic Scheduling in Pentium Pro

° PPro doesn't pipeline 80x86 instructions

° PPro decode unit translates the Intel instructions into 72-bit micro-operations (- MIPS)

° Sends micro-operations to reorder buffer & reservation stations

° Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations

° Most instructions translate to 1 to 4 micro-operations

° Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations

Limits to Multi-Issue Machines

° Inherent limitations of ILP

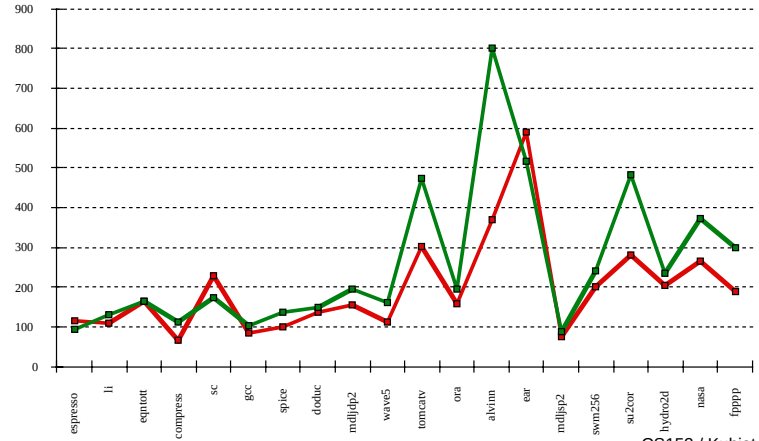
- 1 branch in 5: How to keep a 5-way VLIW busy?
- Latencies of units: many operations must be scheduled
- Need about Pipeline Depth x No. Functional Units of independentDifficulties in building HW
- Duplicate FUs to get parallel execution
- Increase ports to Register File
  - VLIW example needs 7 read and 3 write for Int. Reg. & 5 read and 3 write for FP reg
- Increase ports to memory
- Decoding SS and impact on clock rate, pipeline depth

Limits to Multi-Issue Machines

° Limitations specific to either SS or VLIW implementation
- Decode issue in SS
- VLIW code size: unroll loops + wasted fields in VLIW
- VLIW lock step => 1 hazard & all instructions stall
- VLIW & binary compatibility

Braniac vs. Speed Demon

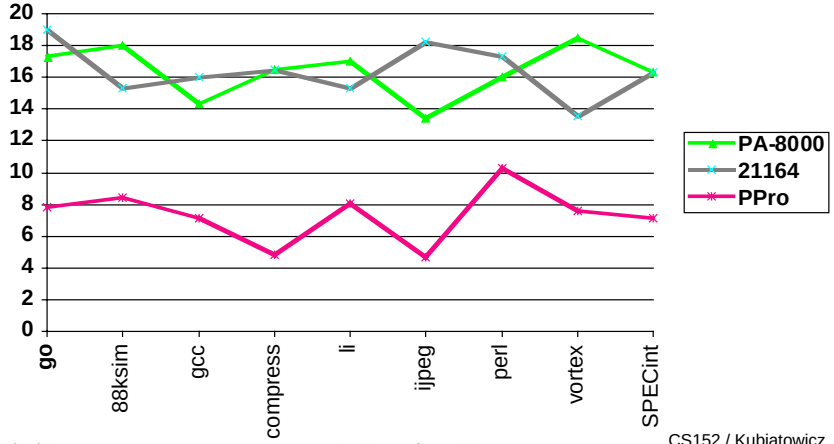
° 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe) vs. 2-scalar Alpha @ 200 MHz (7 stage pipe)



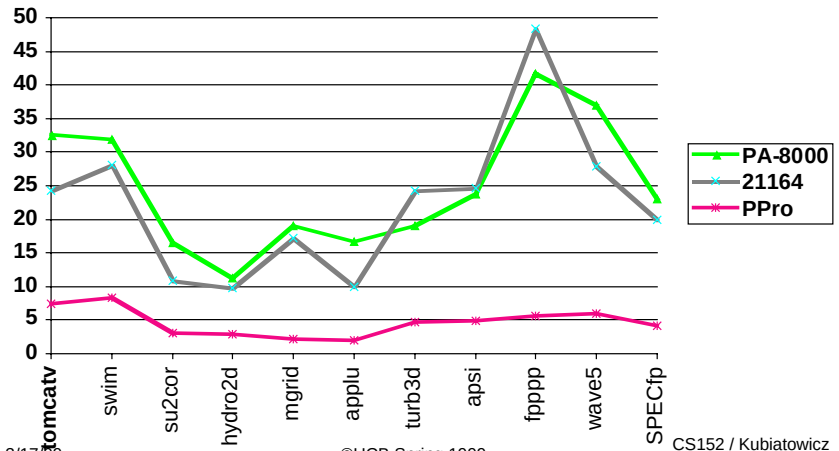
3 Recent Machines

	Alpha 21164	Pentium II	HP PA-8000
Year	1995	1996	1996
Clock	600 MHz ('97)	300 MHz ('97)	236 MHz ('97)
Cache	8K/8K/96K/2M	16K/16K/0.5M	0/0/4M
Issue rate	2int+2FP	3 instr (x86)	4 instr
Pipe stages	7-9	12-14	7-9
Out-of-Order	6 loads	40 instr (µop)	56 instr
Rename regs	none	40	56

SPECint95base Performance (Oct. 1997)



SPECfp95base Performance (Oct. 1997)



Summary

- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- Exceptions in 5-stage pipeline recorded when they occur, but acted on only at WB stage.
- More performance from deeper pipelines, parallelism
- Superscalar and VLIW
  - CPI < 1
  - Dynamic issue vs. Static issue
  - More instructions issue at same time, larger the penalty of hazards
- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead