RISC Architectures:
Fundamentals, Design Alternatives, and Futures?

Includes material taken from presentations by
Professor John Hennessy
and MIPS Computer Systems

Early Statements of RISC Advantages

- **First "RISC" Paper (Patterson and Ditzel, 1980)**
  1. **Implementation Feasibility–single chip implementation will be needed.**
  2. **Design Time–critical in a rapidly growing technology.**
  3. **Speed–faster clock through simplicity and better technology tracking.**
  4. **Better Use of Chip Area–use VLSI for more performance critical functions: caches, faster transistors, or "even pipelining."**

- **Partial explanation in 1983 MIPS paper:**

  "Performance in executing a single sequential instruction stream comes from the ability to execute portions of that stream in parallel. This parallelism may be obtained from either pipelining or multiple functional units."

- **First empirical explanation of RISC in 1985**

What is a RISC Architecture?

Common Properties of RISCs

- **Simple load/store architecture with orthogonal register set.**
- **Simple addressing modes.**
- **Methods for dealing with pipeline delays.**

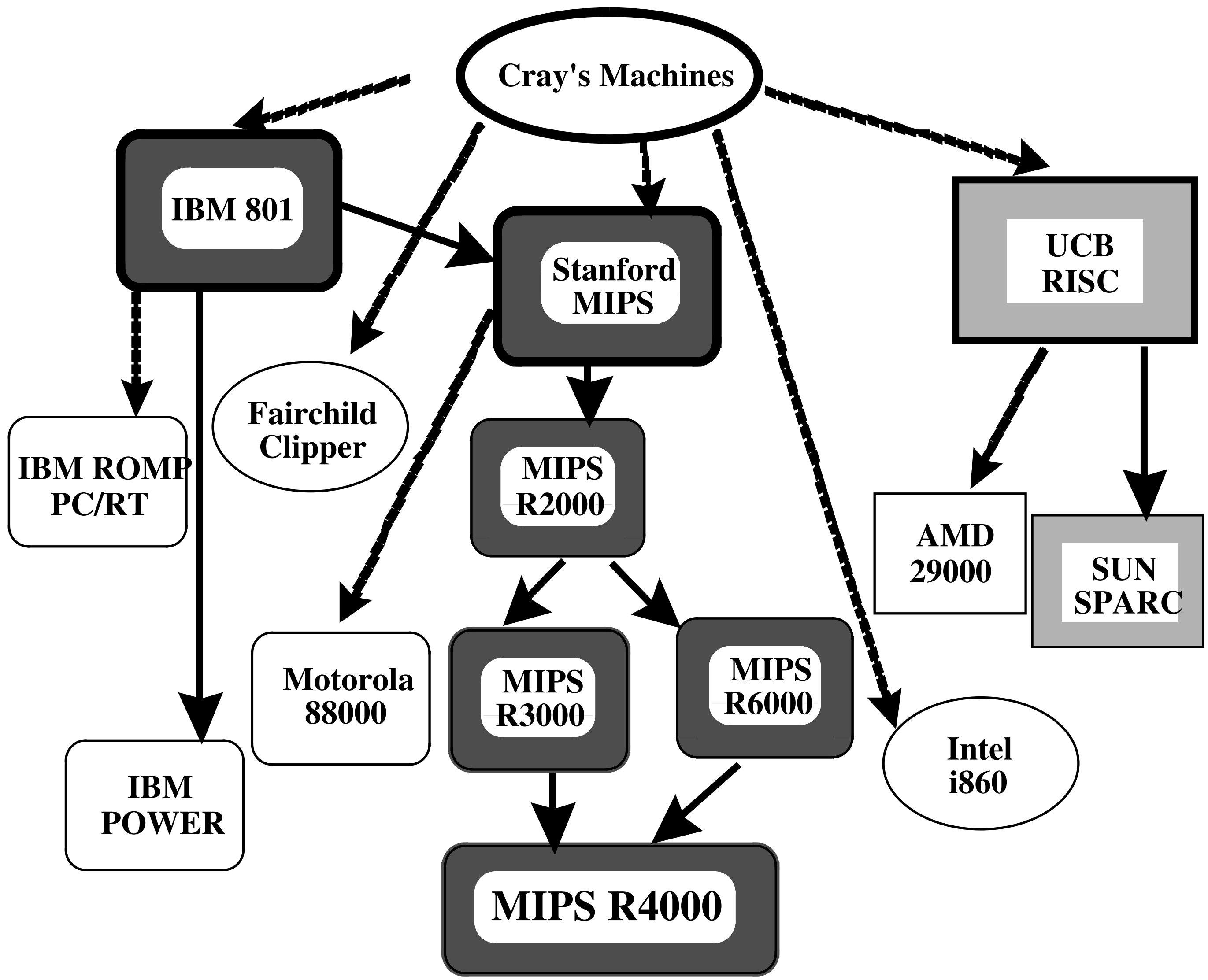
Nonproperties:

- **Small number of instructions.**
- **All instructions are single-cycle.**

Variations:

- **Register number and style.**
- **Branch mechanisms.**
- **Floating point design.**
- **Many implementation differences.**

RISC Genealogy



RISC Principles

GENERAL

- **STREAMLINE HARDWARE, ESPECIALLY COMPLEX FUNCTIONS THAT SLOW CLOCKS OR COMPLICATE PIPELINING.- FOLLOW THE 90% - 10% RULE FOR INCLUDING FUNCTIONS. - MAKE THOSE FUNCTIONS VERY FAST.**

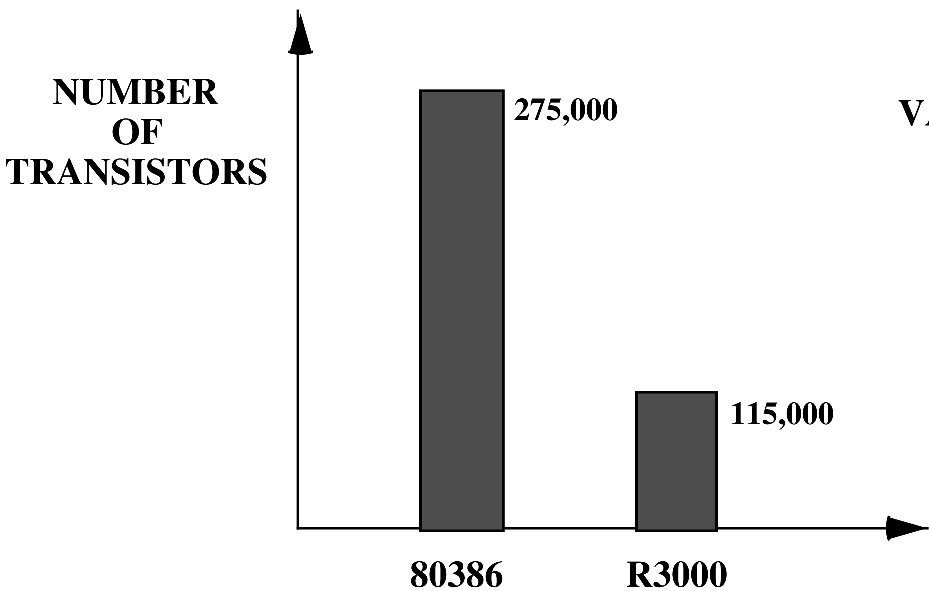
SPECIFIC

- **USE EXTENSIVE SIMULATIONS OF RISC ARCHITECTURES AND HLL PROGRAMS.- RETAIN THOSE FEATURES THAT DEMONSTRATE QUANTATIVELY A NET PERFORMANCE GAIN.- MINIMIZE THE CYCLES PER INSTRUCTION (cpi) PARAMETER**

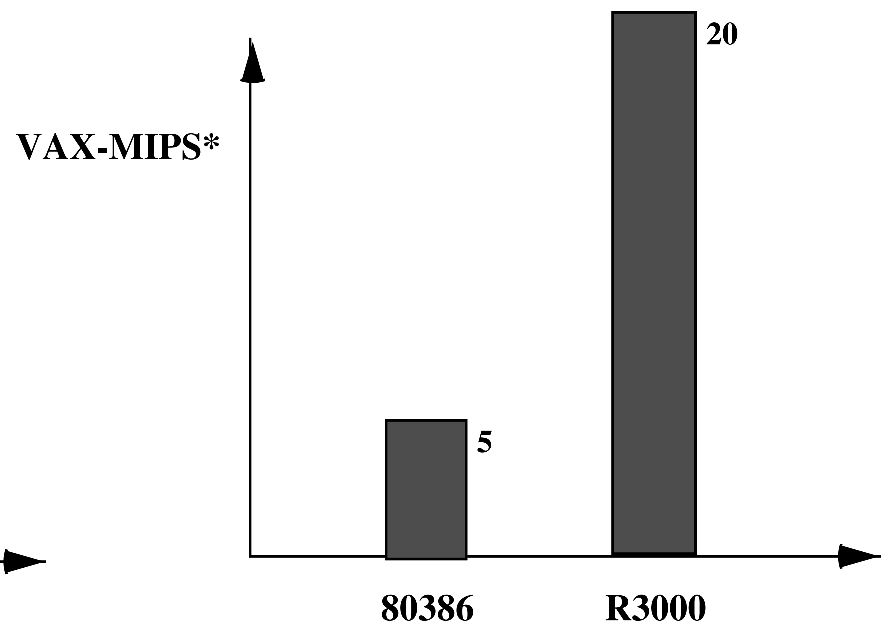
EMPHASIZE SOFTWARE

RISC Benefits

COST



PERFORMANCE



*** BOTH PROCESSORS AT 25 MHz.**

PERFORMANCE EQUATION

TIME/TASK = (INST./TASK) X (CYCLE/INST.) X (TIME/CYCLE)

MEGAHERTZ

NATIVE MIPS

NORMALIZED MIPS*

OVER A LARGE NUMBER OF REAL PROGRAMS

*** Normalized MIPS for a single task, VAX-MIPS or VUPS over a large number of programs.**

A Simple Model for Performance

CPU Performance =

$$\frac{\textbf{Clock Speed}}{\textbf{(Instructions Executed X Cycles per Instruction)}}$$

Clock Speed – Clock Rate of the Machine

Instructions Executed – Dynamic instruction count for some benchmark

Cycles per Instruction – Avg cycles per instruction

Making a Fast RISC Machine

- FAST CLOCK RATE (TIME/CYCLE)
  - VLSI DESIGN
  - LEAN ARCHITECTURE
  - TIGHT INTEGRATION OF THE MEMORY SYSTEM

- LOW CYCLES PER INSTRUCTION (CYCLE/INST)
  - INSTRUCTION SET ARCHITECTURE
  - MEMORY SYSTEM ARCHITECTURE
  - COMPILER TECHNOLOGY

- LOW INSTRUCTION COUNTS (INST/TASK)
  - BALANCED INSTRUCTION SET DESIGN
  - OPTIMIZING COMPILER TECHNOLOGY

Why the RISC Machine Wins

- FAST CLOCK RATE (TIME/CYCLE)
  CLOCK SPEEDS CAN BE MAINTAINED OR
  INCREASED VERSUS CISC ARCHITECTURES

- LOW CYCLES PER INSTRUCTION (CYCLE/INST)
  STREAMLINING THE HARDWARE CAN IMPROVE
  THE cpi BY FACTORS OF FIVE OR MORE

- LOW INSTRUCTION COUNTS (INST/TASK)
  PRICE OF SIMPLIFICATION IS A SMALL INCREASE
  IN THE NUMBER OF INSTRUCTIONS EXECUTED,
  KEPT SMALL BY GOOD OPTIMIZING COMPILERS

What Factors Affect RISC Performance?

First: CPI is affected by:

- **instruction issue ability–accounts for variation in integer CPI**
- **length of pipelines–primarily affects FP**
- **efficiency of instruction scheduling software**

Second: Instruction count affected by:

- **compiler effectiveness–dominates instruction set differences**
- **instruction set differences**

Third: Clock rate affected by:

- **technology available–half generation differences**
- **system implementation assumptions–what cache RAMs, virtual cache versus physical, MP support**

Where does CPI advantage come from?

Lower CPI comes from pipelining.

- **Pipelining exploits instruction-level parallelism.**
- **For today’s RISC machines about 4-fold parallelism is used with pipelines of depth 5.**
- **Non–RISCs achieve parallelism of about 1.**

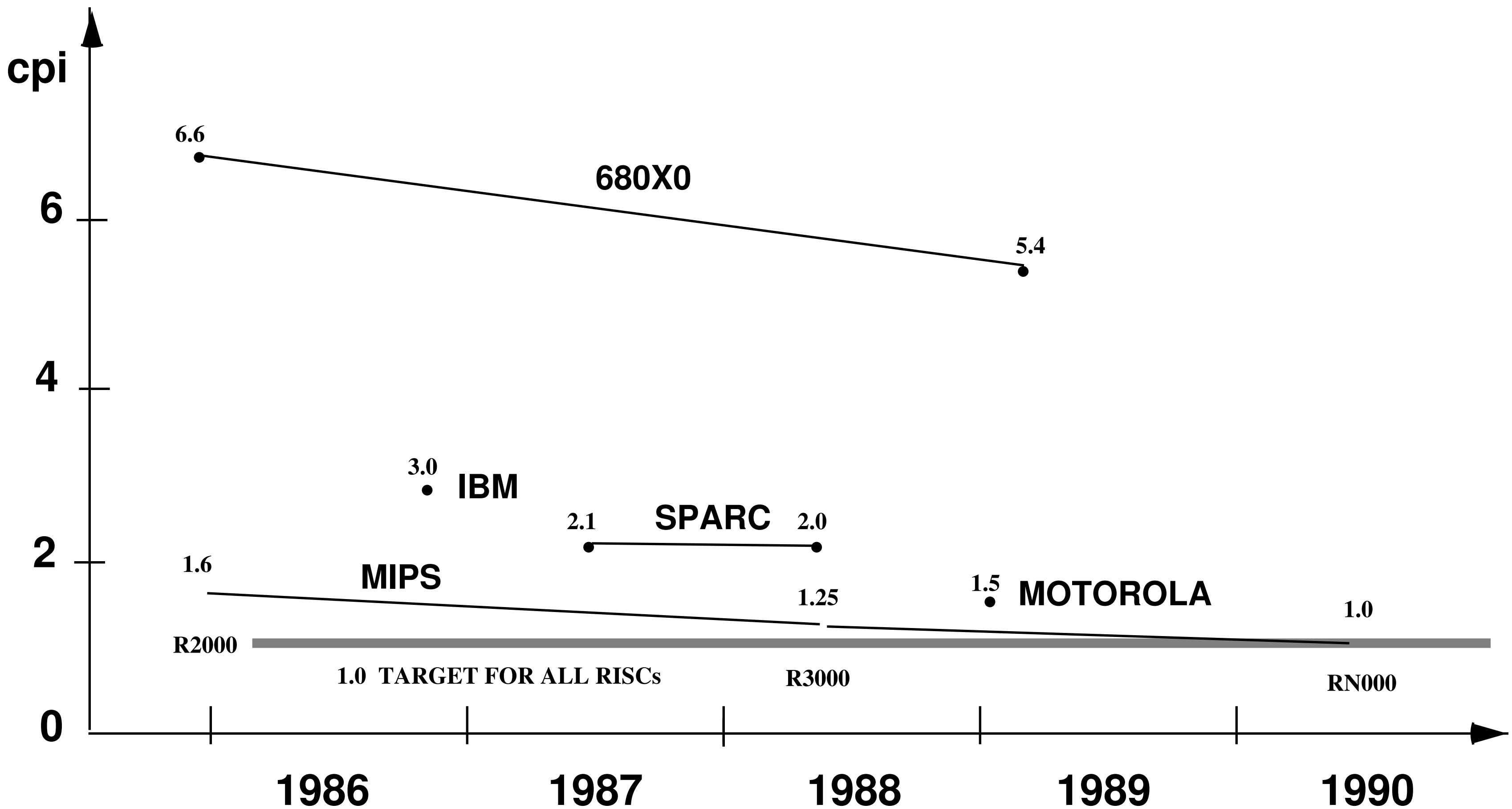
Instruction set density washes out

- **Need 1.8 RISC instructions per VAX instruction, so performance advantage is about 4x at equal MHz.**
- **68020 and RISC instruction counts are about the same, so advantage is about 4x in this case as well.**

Native Instruction Execution Rates

Machine	Clock Rate (MHz)	CPI	Native mips
VAX 11/780	5	10	0.5
VAX 8550	22.2	9	2.5
Moto. 68020	16	7	2.3
Fairchild Clipper	33	6	5.5
Stanford MIPS	4	2	2.0
IBM RT-PC	6	3	2.0
MIPS R2000	16	1.6	10.0
SUN 4/280 (SPARC)	16	2.1	7.6
Motorola 88000	20	1.6	12.5
MIPS R3000251.320			

Cycles per Instruction (MHz/VUP)



Costs of a RISC Approach

- High performance RISC machines need lots of memory bandwidth.
- This is true for all high performance machines - RISCs are not appreciably different in overall bandwidth requirements per mips.
- The fastest RISC machines need up to 2 data words per cycle.
- This bandwidth must be supplied by caches (external or internal/external).
- RISC machines usually have a higher instruction bandwidth requirement arising from less dense instructions and encodings.
- RISCs often have a lower data bandwidth requirement.
- High speed static RAMs make the necessary caches possible and cost effective.

The VAX Story

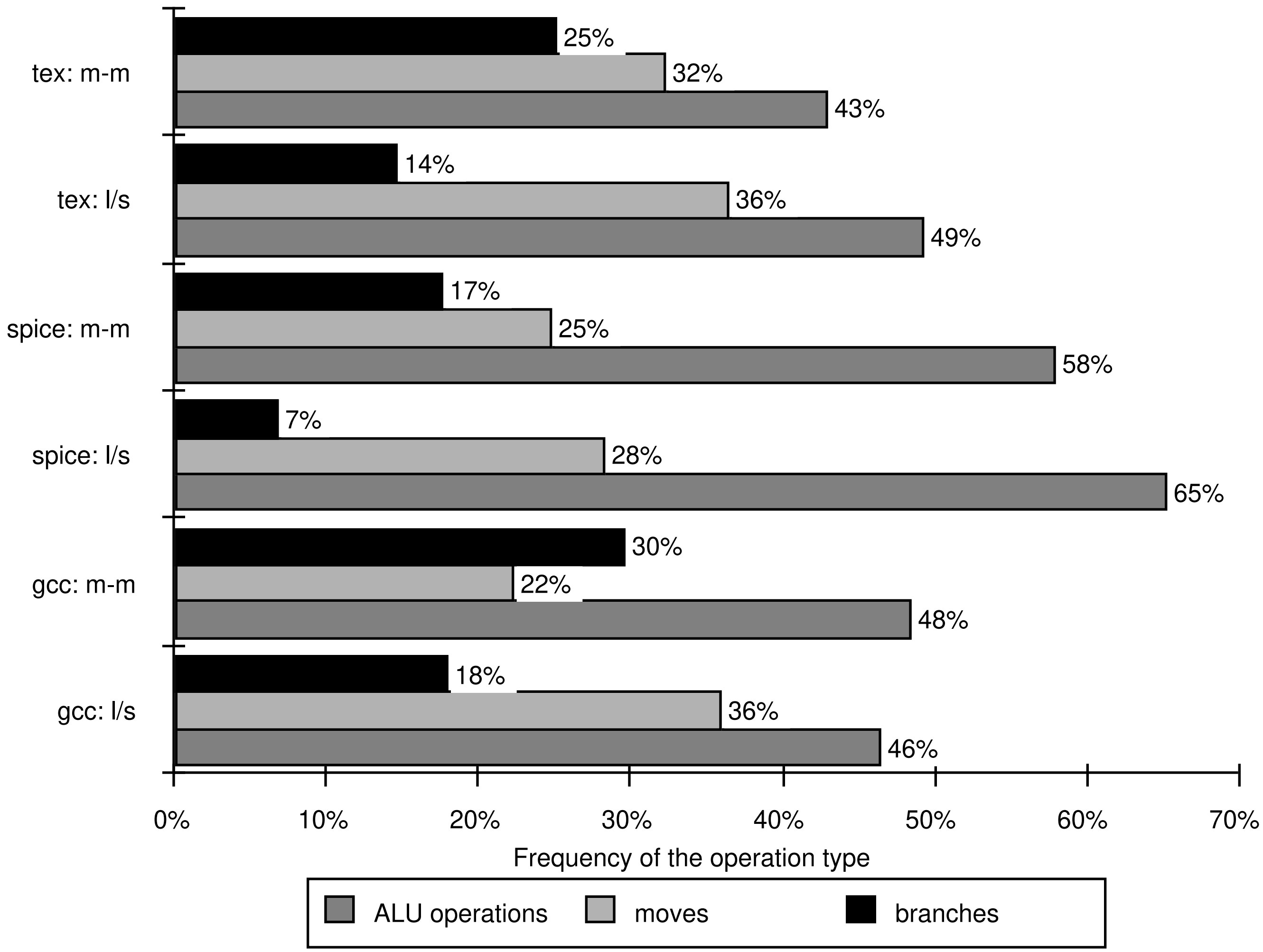
Vax implementations:

Model	Tech.	Clock	Cycles/instr	Native MIPS
11/780	TTL	5 MHz	10	0.5
MV-II	nMOS	~ 6 MHz	12	0.5
8600	ECL	~ 16 MHz		~ 7
8800	ECL	~ 22 MHz		~ 8
CVax	CMOS	16 MHz	10	1.6

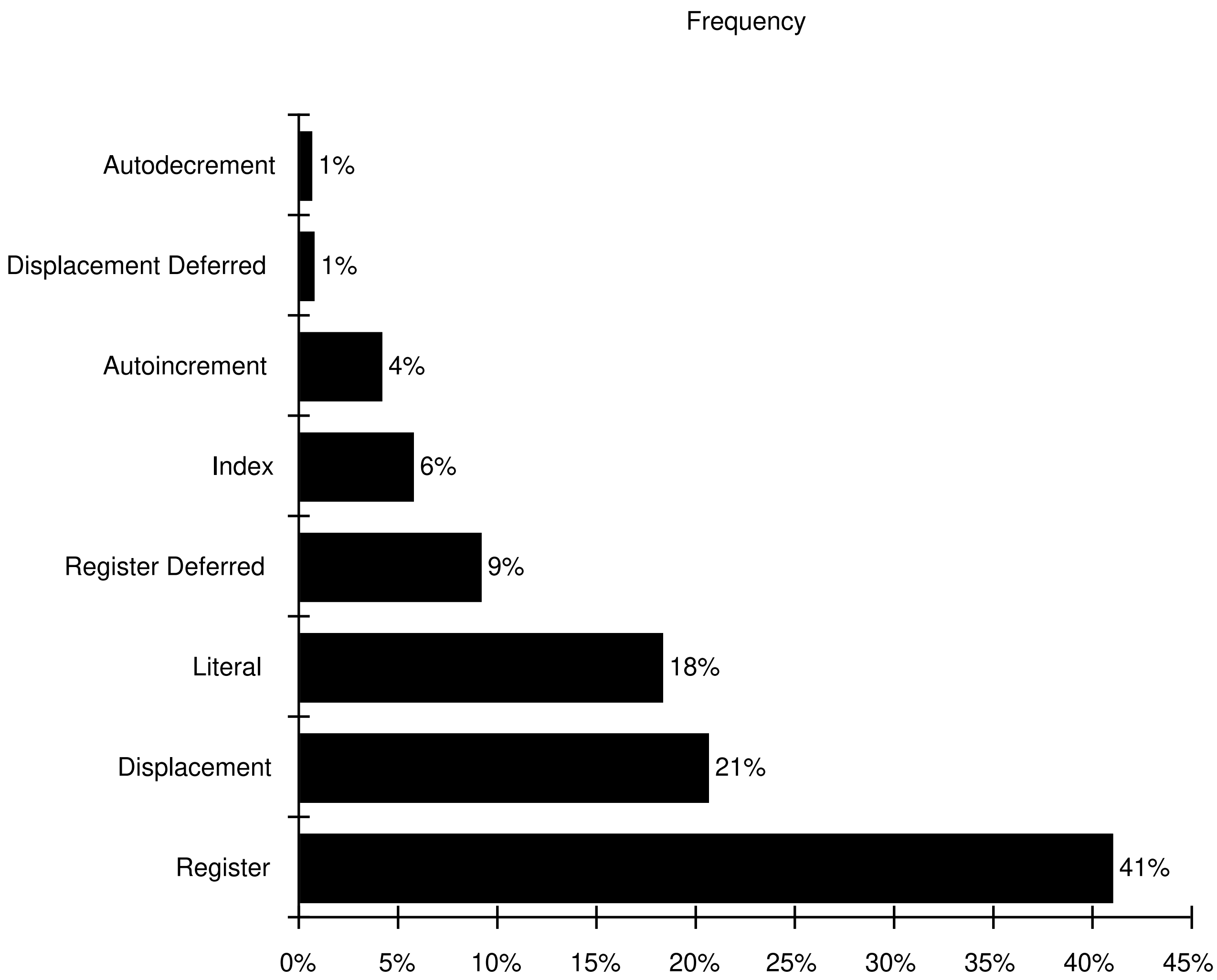
Hard to drive cycles/instruction below 7.

Attempts to do so impact clock speed, size, complexity, and time to implement - dramatically !

Instruction Set Usage–VAX and MIPS



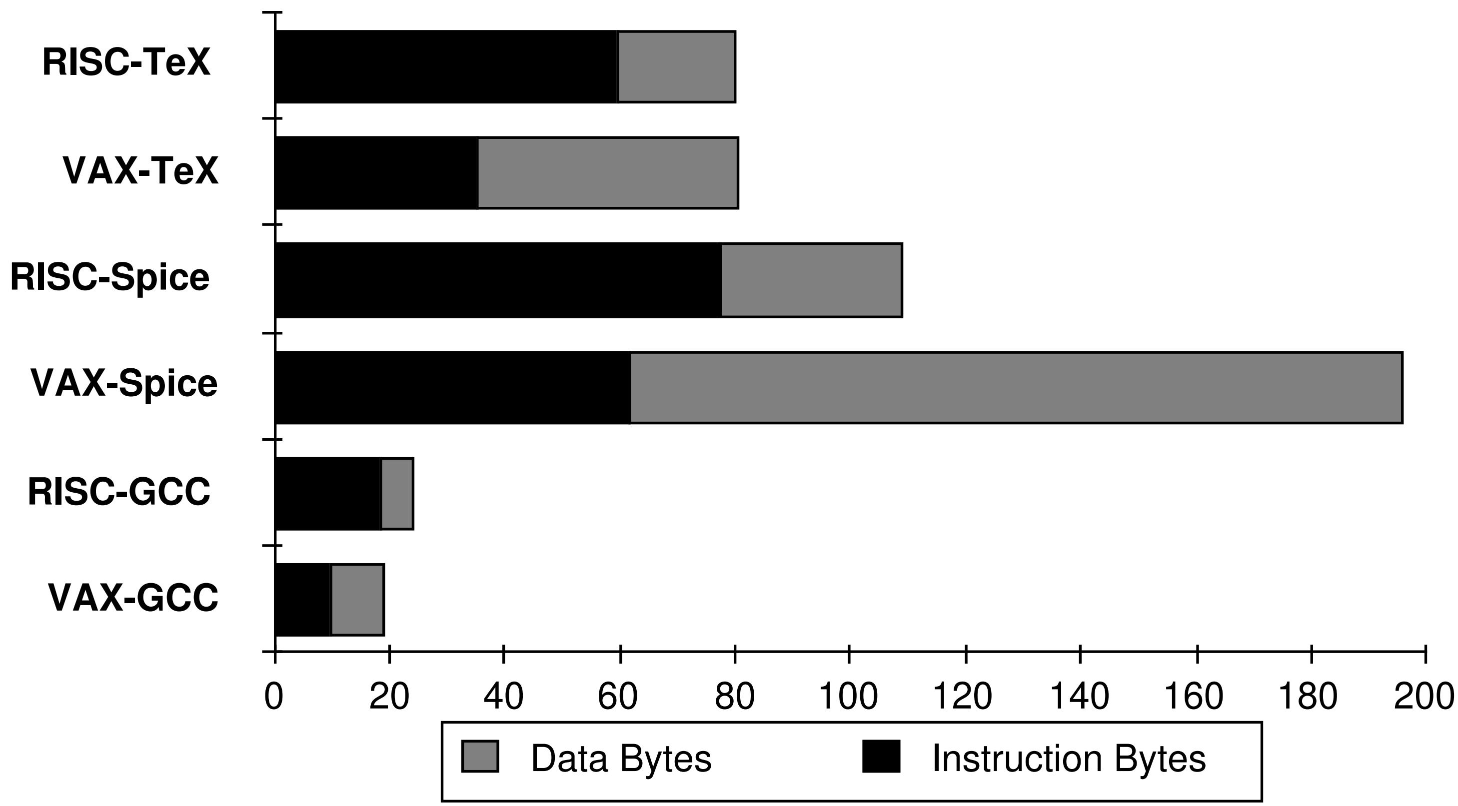
Addressing Mode Usage



VAX vs RISC Memory Bandwidth

- High performance RISC machines need lots of memory bandwidth.
- This is true for all high performance machines - RISCs are not appreciably different in overall bandwidth requirements per mips.
- The fastest RISC machines need up to 2 data words per cycle.
- This bandwidth must be supplied by caches (external or internal/external).
- RISC machines usually have a higher instruction bandwidth requirement arising from less dense instructions and encodings.
- RISCs often have a lower data bandwidth requirement.
- High speed static RAMs make the necessary caches possible and cost effective.

VAX vs. RISC Memory Bandwidth

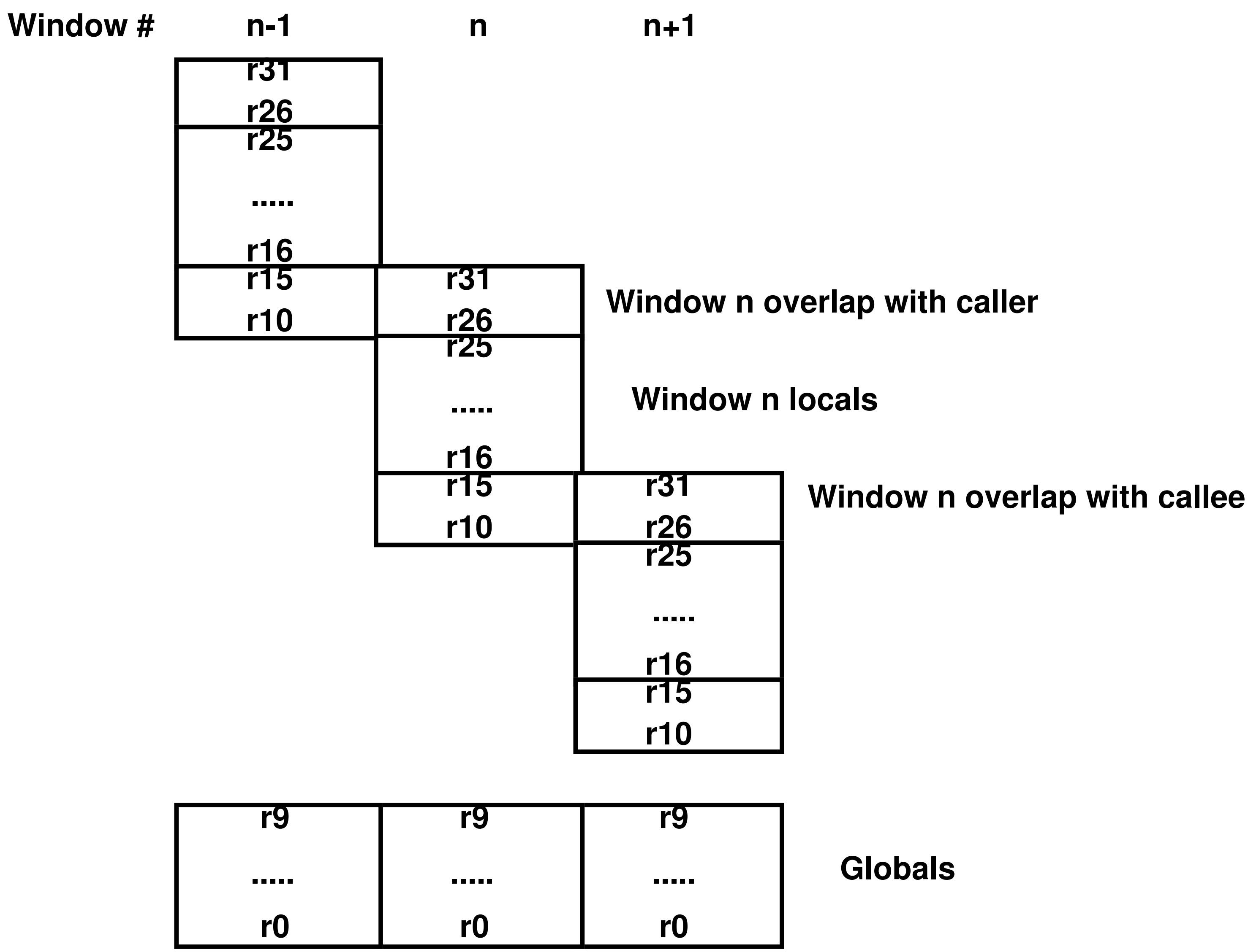


Two Basic RISC Approaches

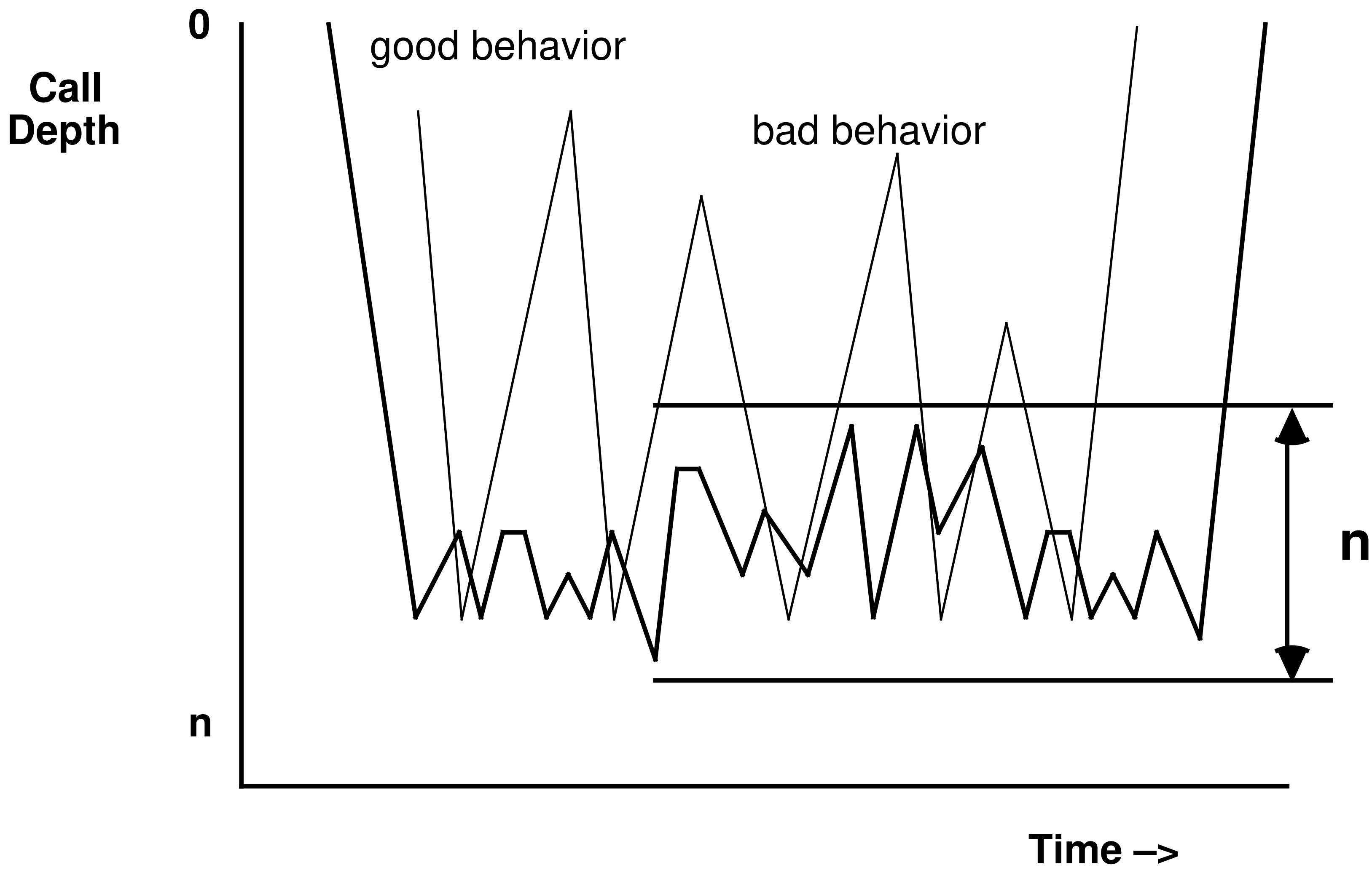
- IBM/STANFORD APPROACH
  - SINGLE SET OF ORTHOGONOL REGISTERS
  - FAST LOAD/STORE; SCHEDULED DELAYS
  - EMPHASIS ON COMPILERS, GLOBAL OPTIMIZATION

- BERKELEY APPROACH
  - MULTIPLE REGISTER SETS ARRANGED AS WINDOWS
  - MOST IMPLEMENTATIONS HAVE RELATIVELY SLOWER LOAD/STORE, 2 OR MORE CYCLES

Register Window Scheme

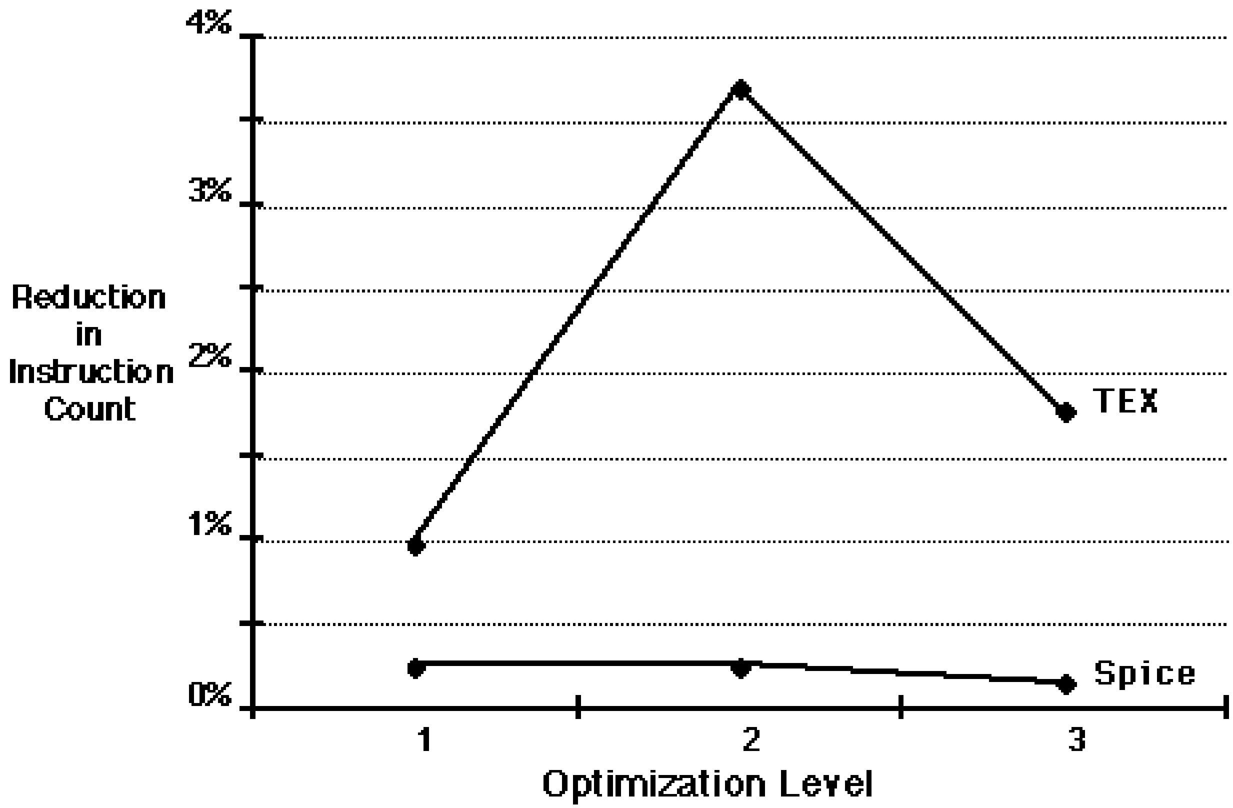


When Do Register Windows Work?



Savings from Register Windows

Measured with MIPS compilers. Correlated with SPARC results.



Comparison of the Two Approaches

- **Performance differences are small:**

    Good register allocator lowers load/store frequency.

    Large fraction of references (~50%) not allocatable anyway.

    Interprocedural register allocation lowers gap to < 5%!

- **Register window scheme is usually more adaptable.**

- **Costs of register window scheme:**

    1. Potential clock cycle increase - large register file is slow

    2. Not much help for floating point applications.

    3. Opportunity cost - consumes lots of silicon

    4. Large register file impacts process switch time.

Minimizing Instruction Execution Count

- **Minimizing the instruction counts can be done by compiler optimization.**

Optimizations Performed	Percent faster
Procedure integration only	10%
Only local optimizations	5%
Local optimizations + register allocation	26%
Global and local optimizations	14%
Local & global optimizations + register allocation	63%
All optimizations + register allocation	81%

- **RISC machines have simple register-register instruction sets.**

- **Bandwidth to registers is much higher than memory.**

- **Conclusion: register allocation can yield major improvements.**

Register Allocation Example

Consider C := A + B:

Load R1,A

Load R2,B

Add R3,R1,R2

Store R3,C

Without registers it takes 4 instructions

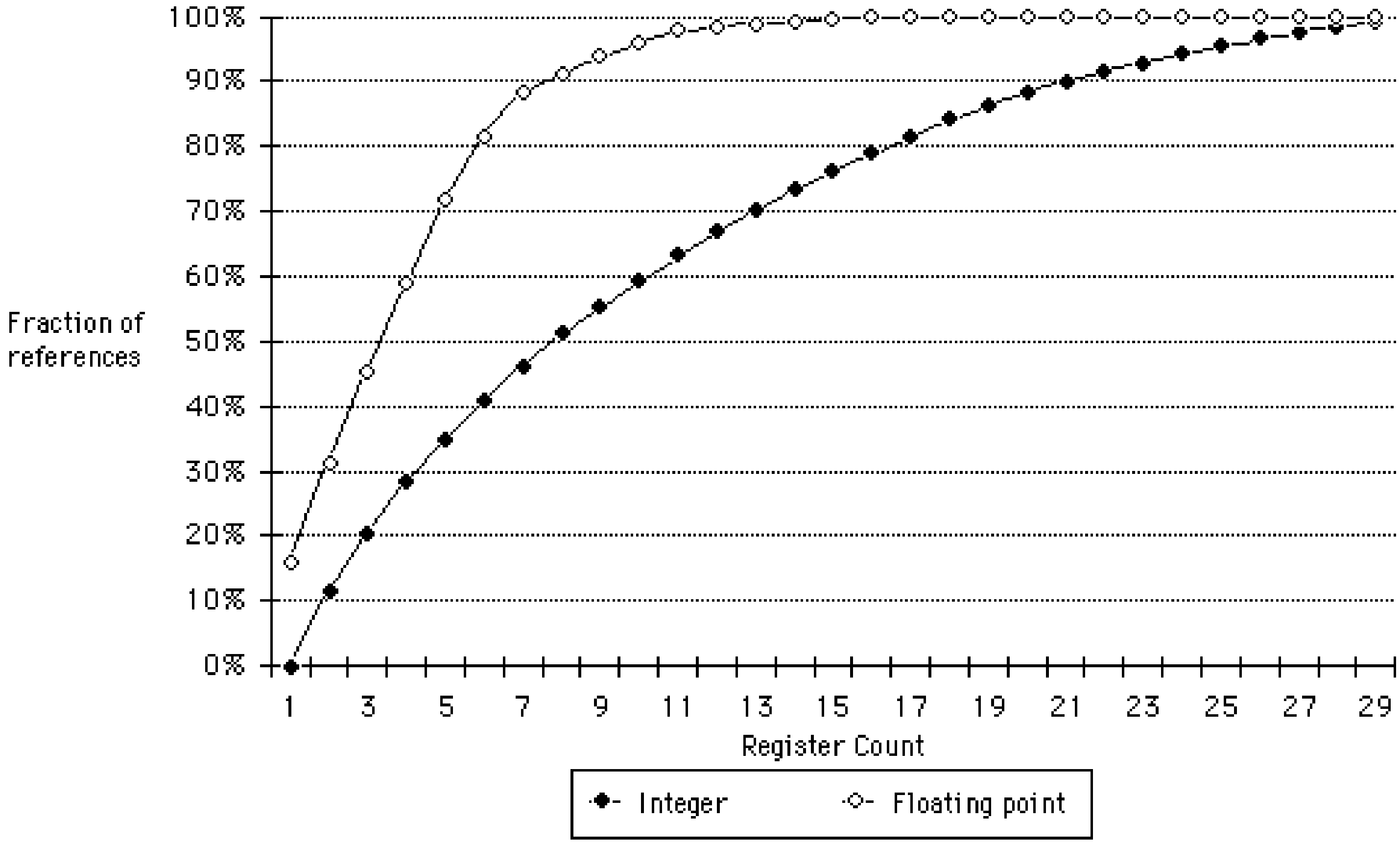
But, with A, B, and C allocated to registers:

Add Rc,Ra,Rb

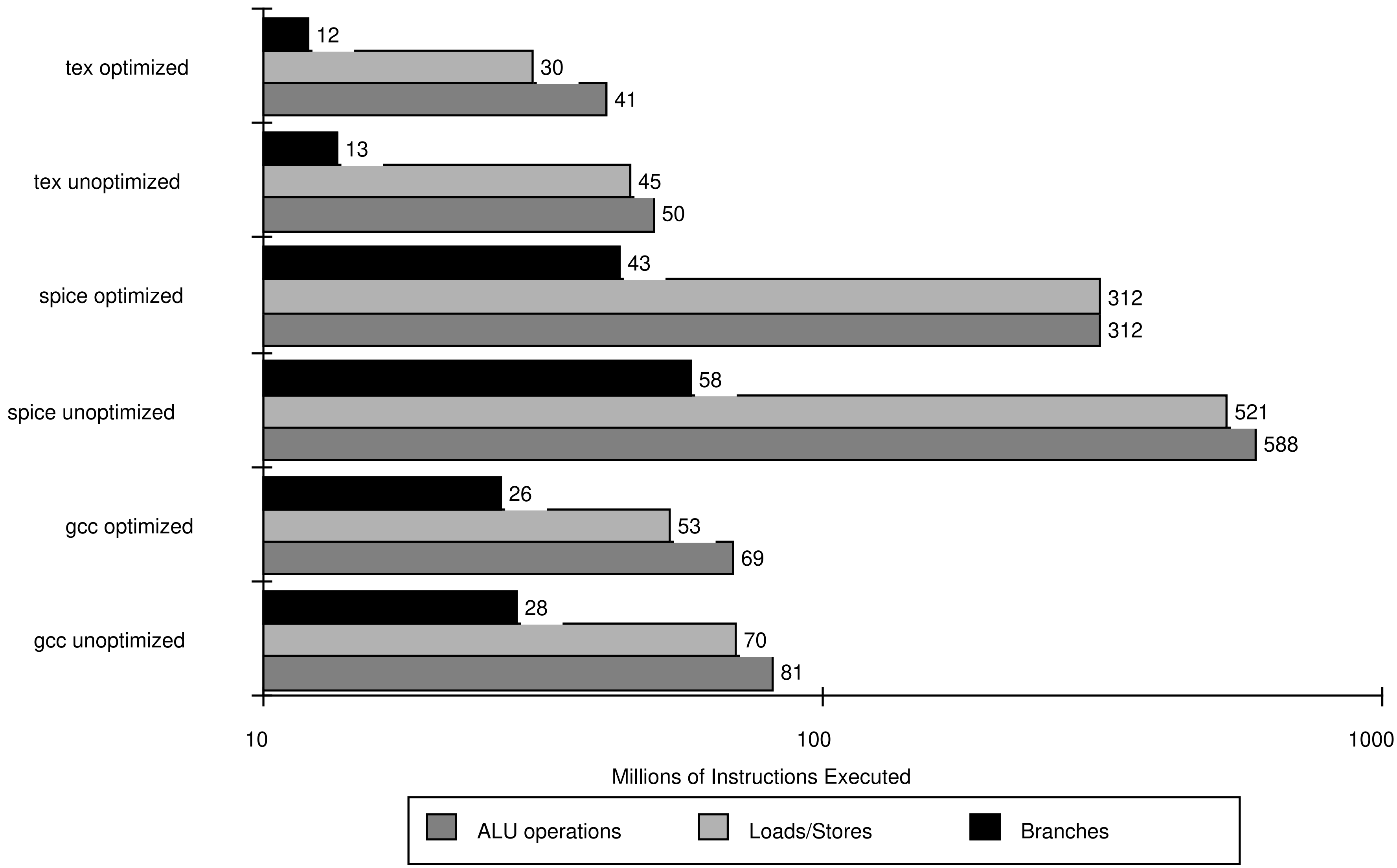
A 4x improvement!!!

- **Modern register allocators use graph coloring algorithm -- sufficient registers are needed to make algorithms work.**

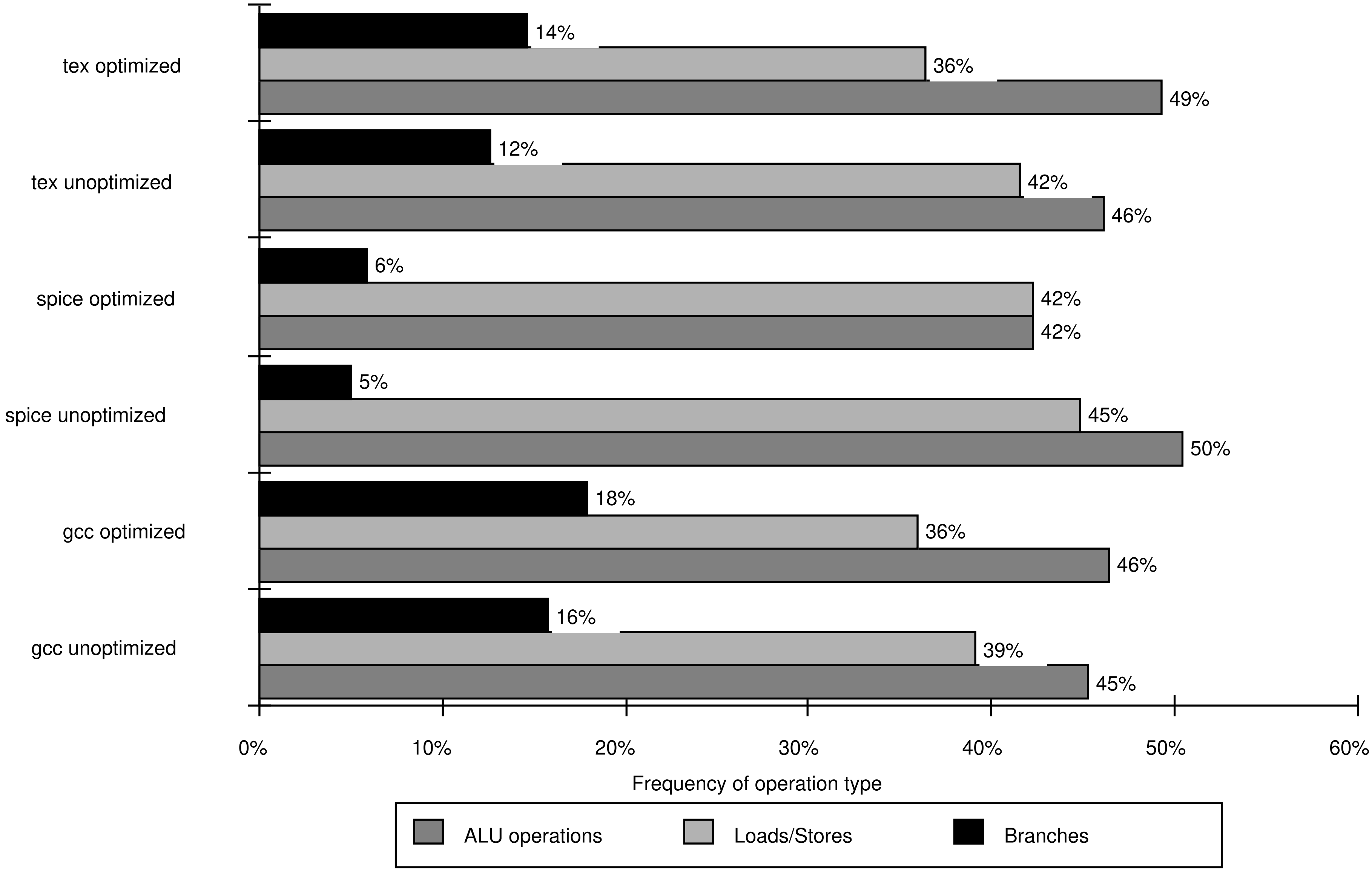
Register Usage



Optimization Affects–Instruction Counts



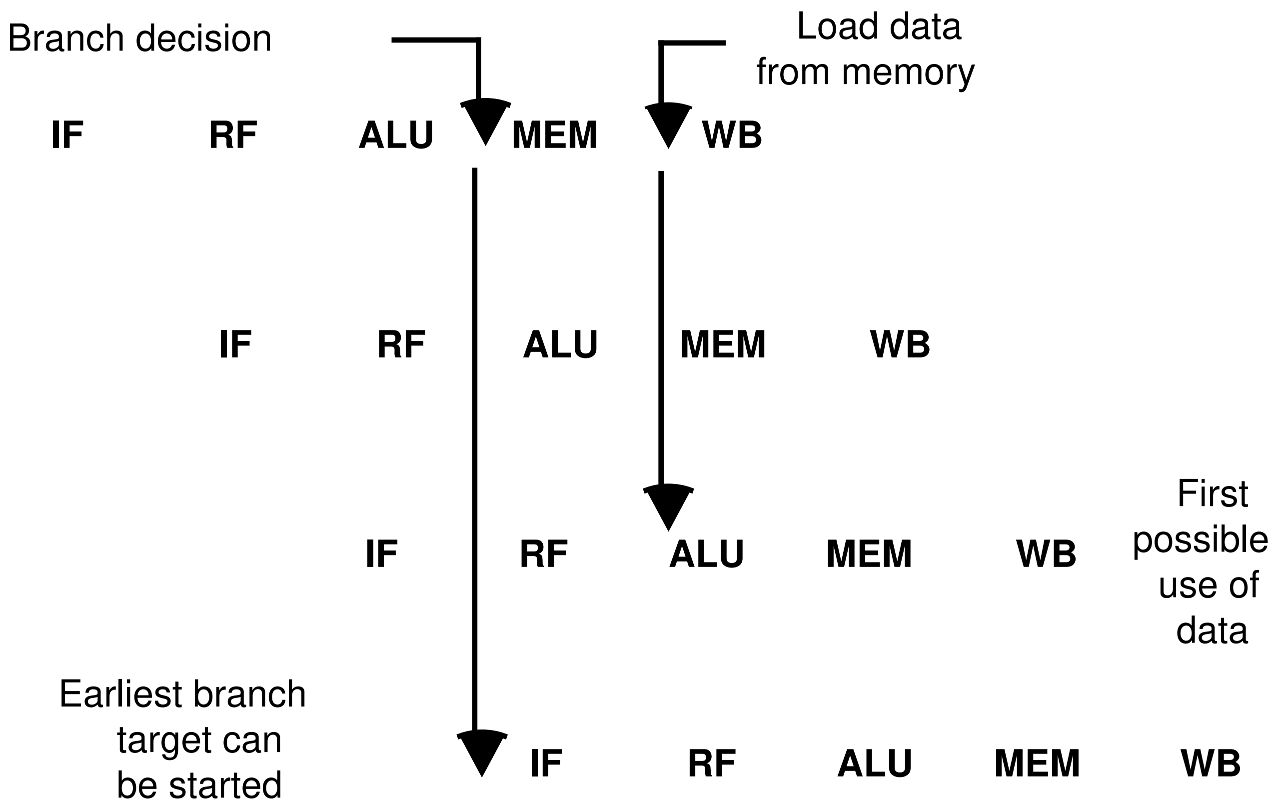
Optimization Affects–Instruction Frequencies



Minimizing the Cycles per Instruction

- High performance RISC machines strive to achieve one-cycle execution for all instructions.
- Pipelining allows the machine to initiate an instruction on each clock tick.
- Efficient pipelining is perhaps the most important property that RISC architectures have.
- Not all instructions can be completed in one clock cycle – some instructions have a latency in their effect.
- Branches, loads/stores, and FP ops have this property.
- Latency must be removed to keep single-cycle execution.

Typical RISC Pipeline



IF - Instruction fetch

RF - Register Fetch
ALU - ALU operation
MEM - Memory reference
WB - Write Back to registers

This pipeline has :

Branch latency of 2
Load latency of 1

Instruction Costs and Latencies

- The minimum cost for an instruction depends on structural hazards and instruction capabilities:
  - A structural hazard prevents some instruction from issuing on every clock.
  - Some architectures may require multiple instructions for the same function.

- Structural hazards can occur for branches and memory references.

- They are caused by:
  - the number of memory pipelines (1 versus 2),
  - how branch conditions are determined.

Architecture and Compiler Goals

1. Minimize the cycle count for the instruction (eliminate structural hazards).

2. Minimize the delay length for the instruction (affected by pipeline design).

3. Eliminate delays that remain with software pipeline scheduling

Loads and Stores

- Cycle count determined by cache structure and addressing modes.

- For almost all existing RISC machines, delay length = 1.

- Costs for load/store with addressing mode = base register + offset:

Machine	Load	Store	
MIPS R2000	1	1	(2 memory pipes; write through cache)
SUN 4	2	3	(1 memory pipe; write back cache)
Berkeley RISC	2	2	(1 memory pipe)
MIPS-X	1	1	(2 memory pipes; delay restriction on store-successor not load or store)
AMD 29000	2	2	(must add offset)

Optimizing the load delay

- Consider the code for C := A + B; F := D + E:

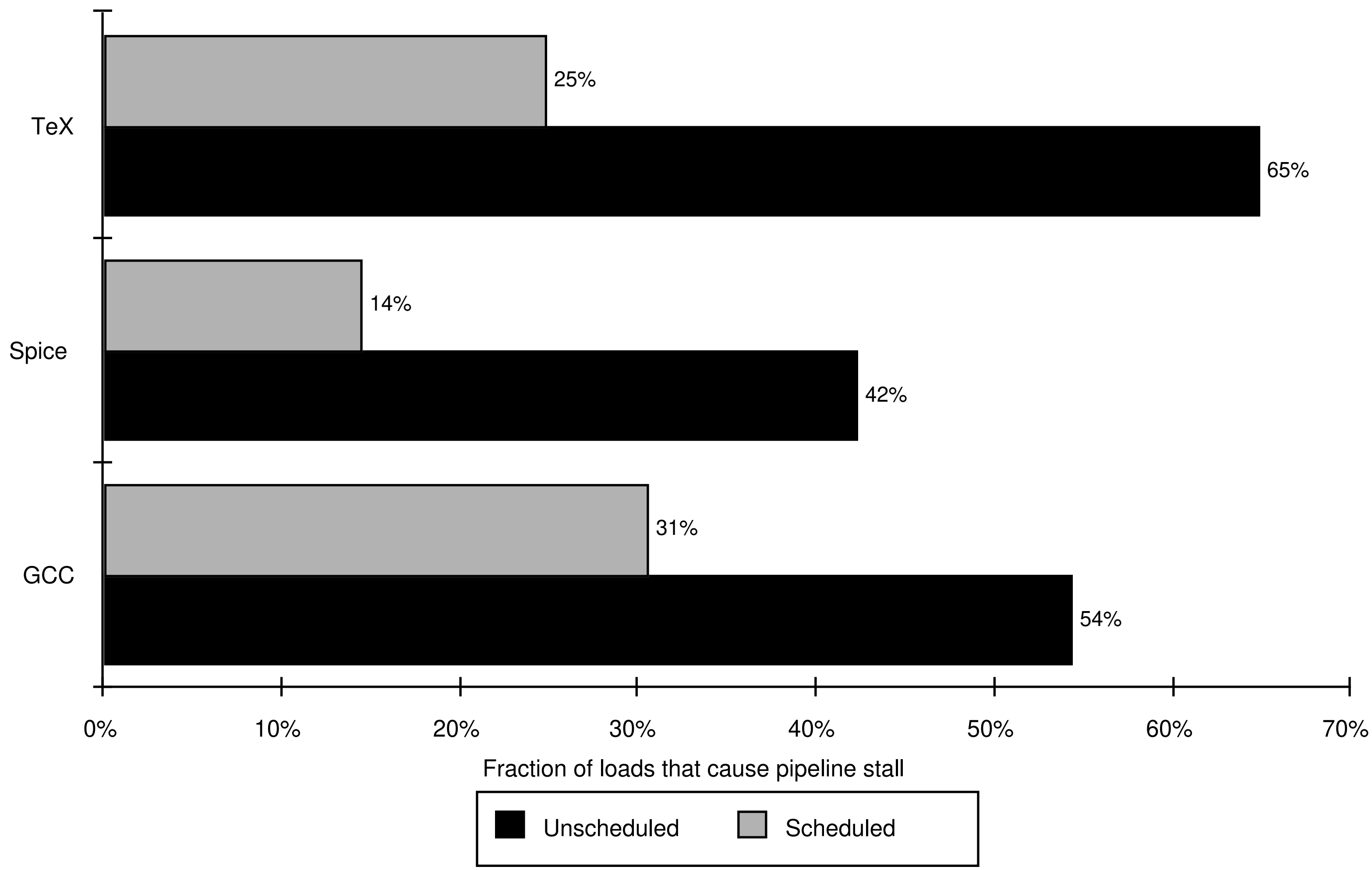
```
Load R1, A
Load R2, B
Add  R3,R1,R2 -- this instruction stalls
Load R4, D
```

- An alternative code sequence is (delay length =1)

```
Load R1, A
Load R2, B
Load R4,D
Add R3,R1,R2 -- no stall since R2 ready
```

- Instruction in the delay slot must be independent of load.

Effectiveness of Load Delay Scheduling



Different Branch Architectures

- **Branch condition can be set by:**

  Full compare in branch: Stanford MIPS and MIPS-X

  Limited compare in branch: MIPS R2000, Moto. 88000

  Controlled Condition codes: Berkeley RISC, SPARC, CRISP

  Condition register: IBM 801

- **Branch type:**

  Ordinary delayed: MIPS, RISC, MIPS R2000, Moto 88000

  Optionally delayed: IBM 801

  Cancelling: HP Precision, MIPS-X, SPARC

  Hardware prediction: CRISP, AMD 29000

Delayed and Cancelling Branches

- **With a delayed branch, one or more instructions after the branch are always executed.**

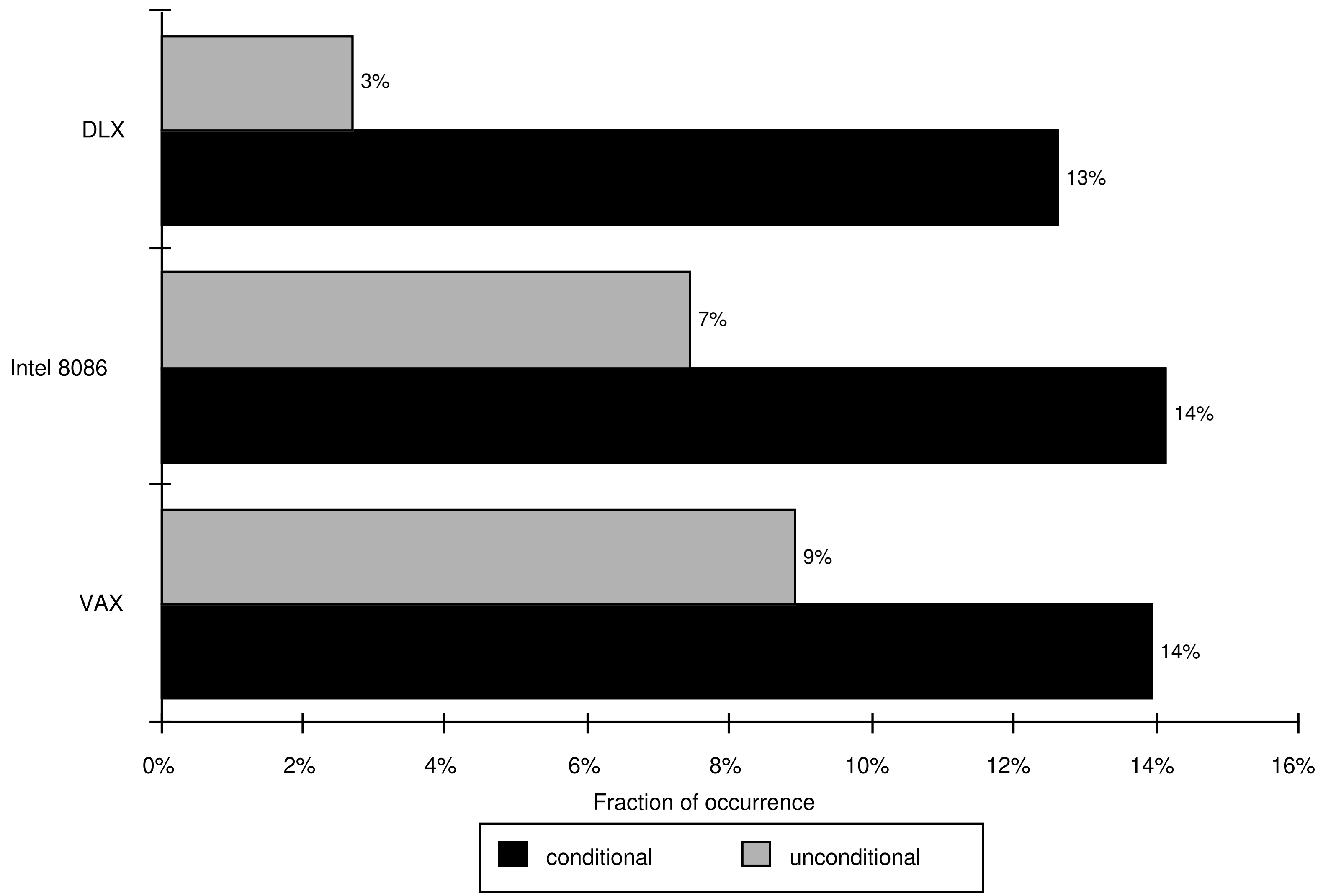
- **A cancelling branch allows the instruction in the delay slot to be cancelled if branch is taken/not taken.**

- **Example: A := B + C; if B = 0 goto L. Usual code is:**

```
    Add  Ra,Rb,Rc
    Beq Rb,0,L
    ...........  --  in delayed branch these instruction(s) are executed        ...........
  L:execution continues here
```

- **In a cancelling branch the instructions in the delay might be cancelled.**

What Types of Branches Occur?



Branch Costs and Delays

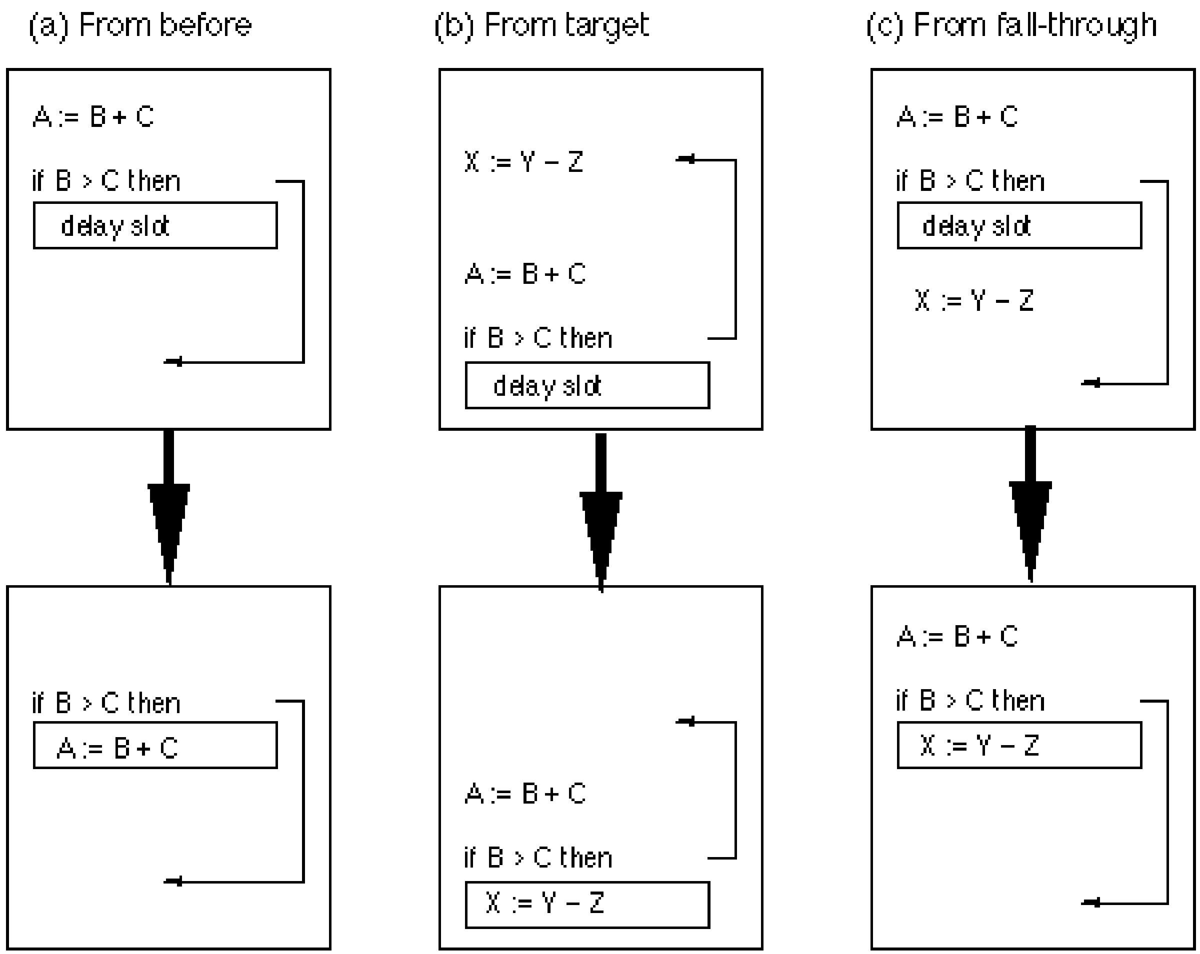
- Branch costs and delays vary in machines:

Machine	Branch cost	Delay length
Stanford MIPS	1 cycle	1 cycle (pipeline 3 deep)
MIPS R2000	~1.1 cycle	1 cycle (extra compare 5%)
Berkeley RISC	2 cycles	1 cycle (must set CC)
SUN 4	1.8-3 cycles	1 cycle (set CC; miss penalty)

- Cancelling branches increase the pool of safe instructions and are critical if branch delay > 1.

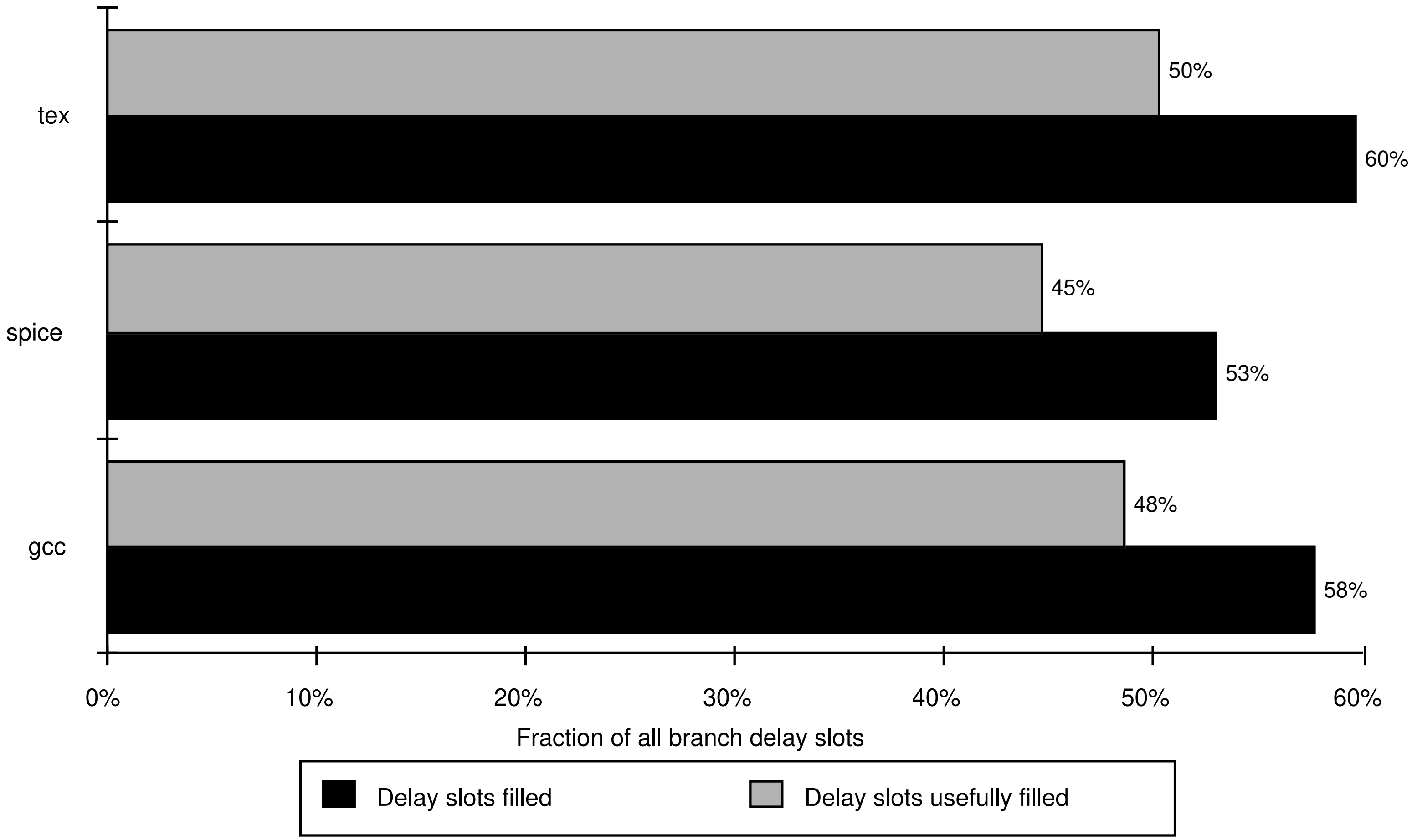
- Scheduling a branch delay slot may require static branch prediction

Scheduling a delayed branch



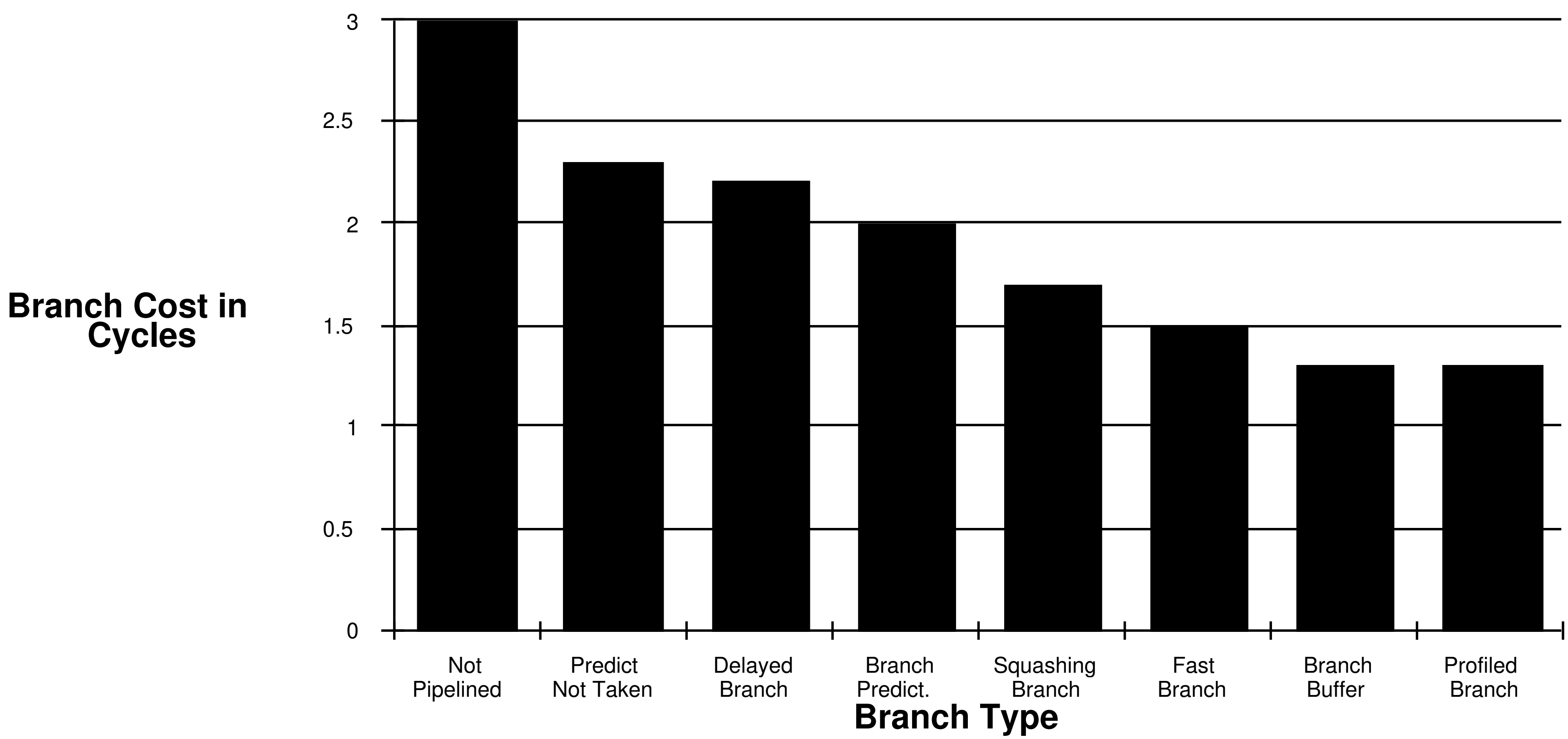
Effectiveness of Simple Delayed Branch

- Filled – something was put in slot
- Usefully Filled – when compiler guessed right

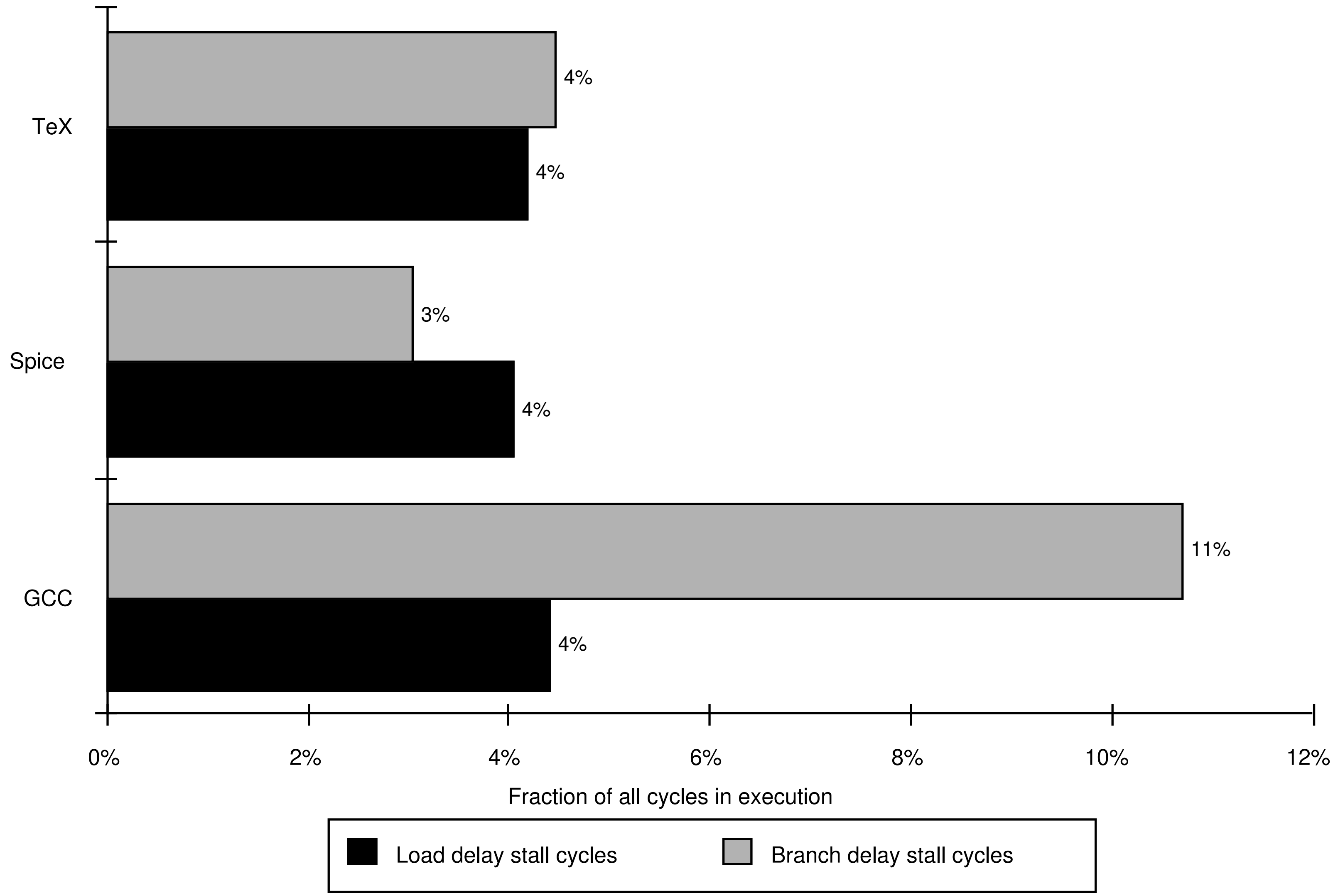


Branch Cost in Cycles for Various Schemes

- Full compare and branch = 3 cycles.
- Perfect branch = 1 cycle.



Total Affect of Integer Pipeline Stalls



Fast Floating Point on a RISC

Challenge: integrate inherently multicycle FP ops into a RISC machine.

Approach: FP in a separate coprocessor with its own FP regs.
Load/store coprocessor registers from memory;
CPU must do address calculation.

Difficulty: Can't move data back and forth between CPU and FP

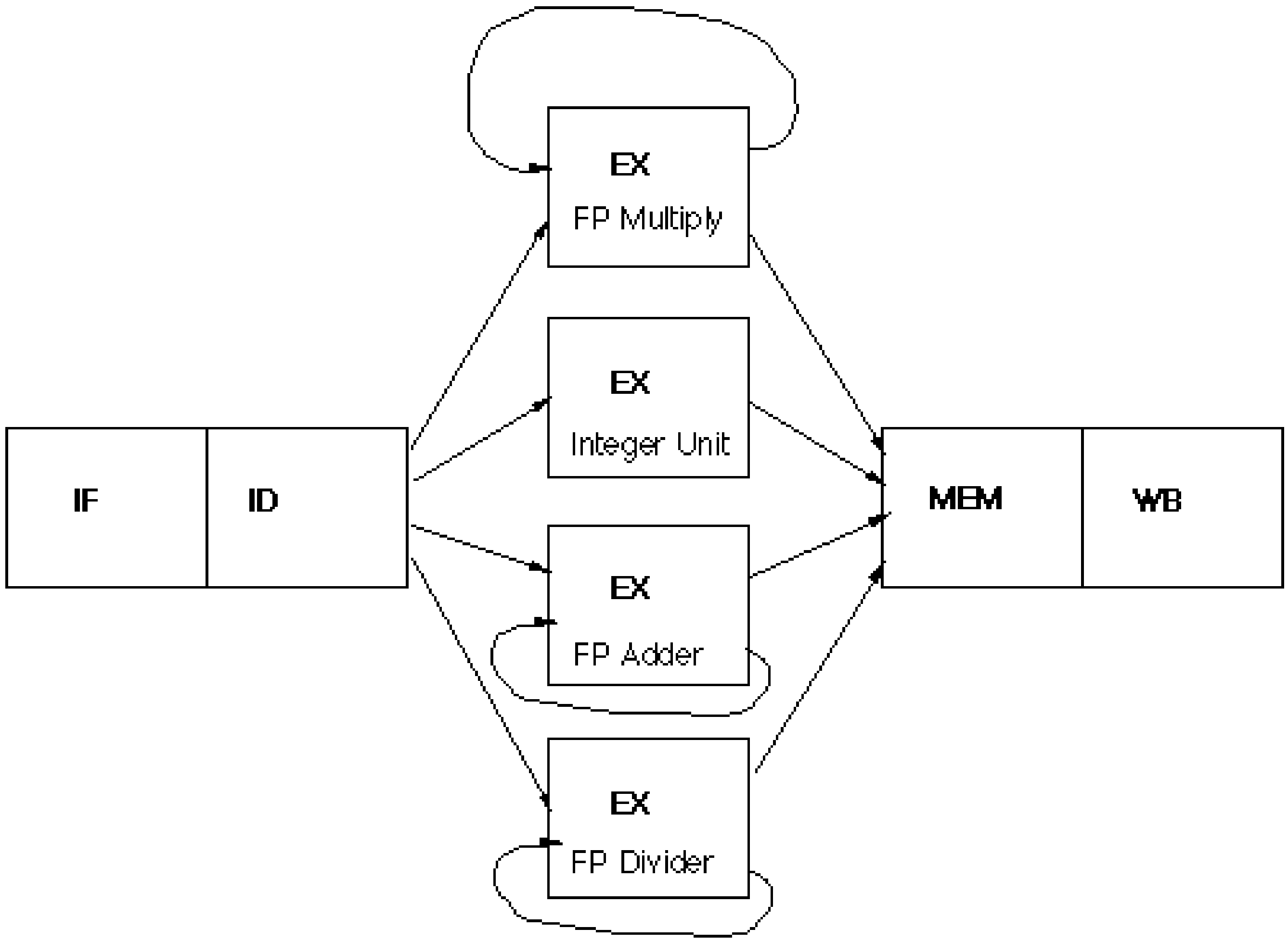
Solution: FP watches CPU bus and grabs data on proper cycle.

Difficulty: FP and CPU pipeline are very different.

Solution: Run FP as autonomous in parallel with CPU.
Make multicycle ops into delayed ops and schedule
Short latencies critical for full pipeline.

FP Pipeline

• **Leads to out-of-order completion.**



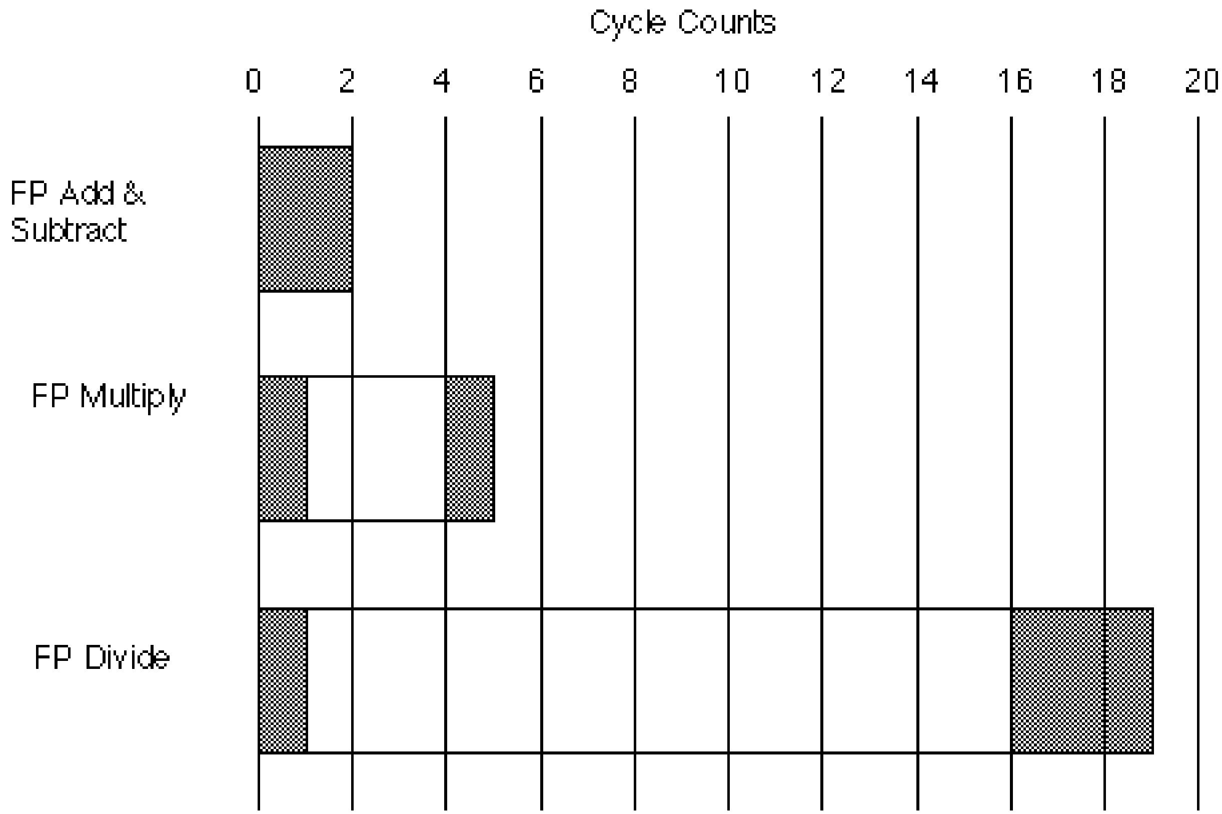
Out-of-Order Completion in the FP Unit

- **How can we retain precise interrupts?**

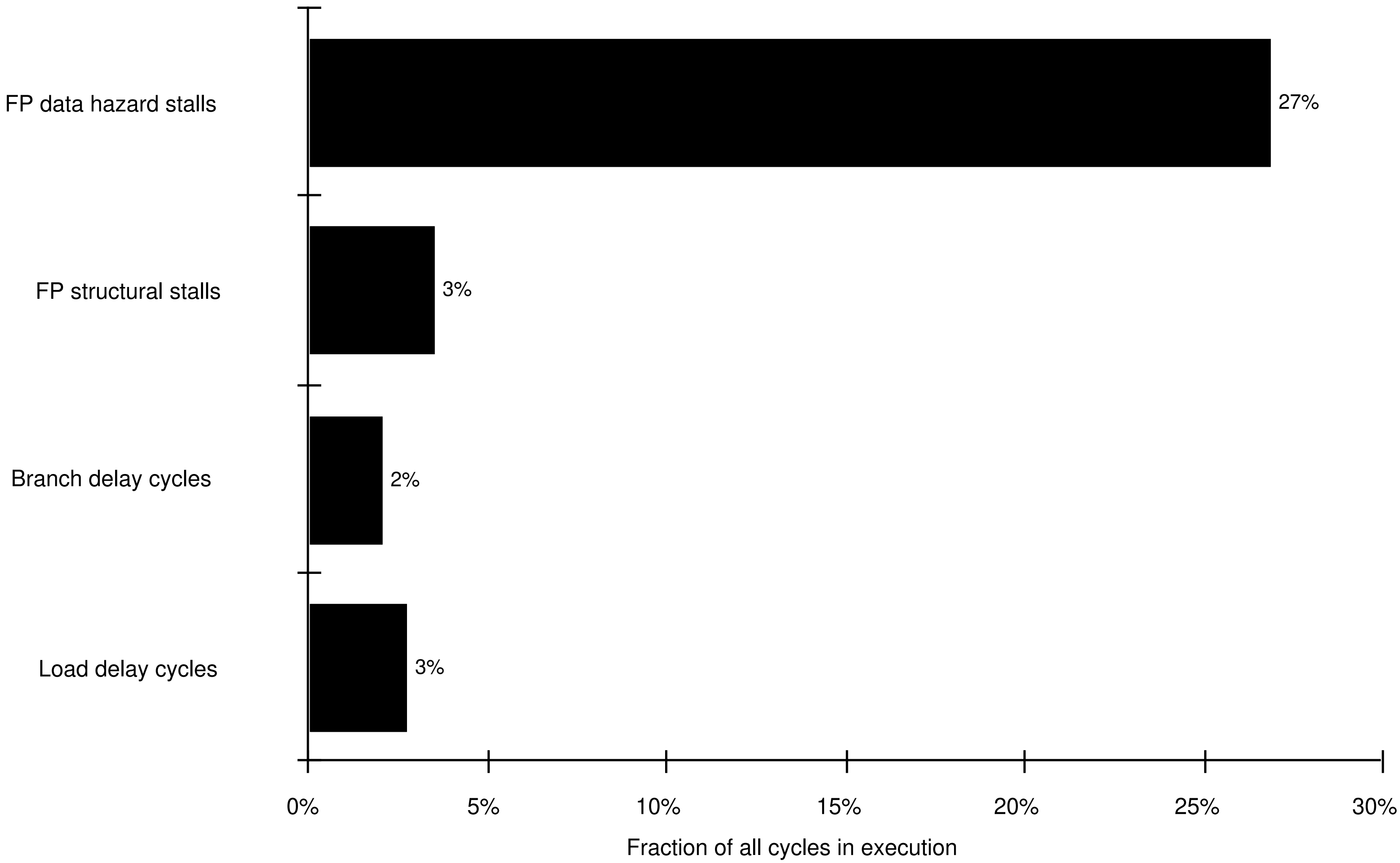
- **Keep integer unit partially synchronized.**

- **Remaining problems arise from FP exceptions; approaches:**
  1. **Very deep buffers for results–impractical.**
  2. **Keep queue of PCs of all pending FP operations–complete partial operations in OS to obtain clean interrupt state. SPARC approach.**
  3. **Detect possible FP exceptions early–if possible stall instruction completions. MIPS approach.**
  4. **Partially imprecise interrupts. Intel and Motorola approach.**

MIPS R2000/3000 FP Unit



FP Pipeline Stalls



Maximizing Clock Cycle

• To maximize clock cycle shorten critical paths in processor.

• RISC execution engine is compact => critical path will be in:

1. Memory access
2. Instruction decode and dispatch
3. Exception handling and pipeline control

Memory access time - instructions and data must be fetched:

1. Minimize memory mapping delay
   Virtual cache or high-speed integrated TLB
2. Prefetch instructions: fixed size simplifies.
3. Early data address calculation: single addressing format
4. Minimize cache overhead: organization and hit detection.

Instruction fetch/decode

Minimized by simple instruction encoding; typical (R2000):

- I-Type Instruction

6	5	5	16
opc	rs	rt	immediate

Loads and Stores: bytes, words, half-words

ALU immediate (rs <- rt op immediate)

Branch (rs=rt,rs positive)

- R-Type Instruction

6	5	5	5	5	6
opc	rs	rt	rd	re	func

Register-register ALU operations:

Func encodes the datapath operation: Add,Subt, ...

For shift instructions the re field encodes the shift amount

- J-Type Instruction

6	
opc	target - absolute address

Jump and Jump and link to an absolute address

Pipeline control and interrupt handling

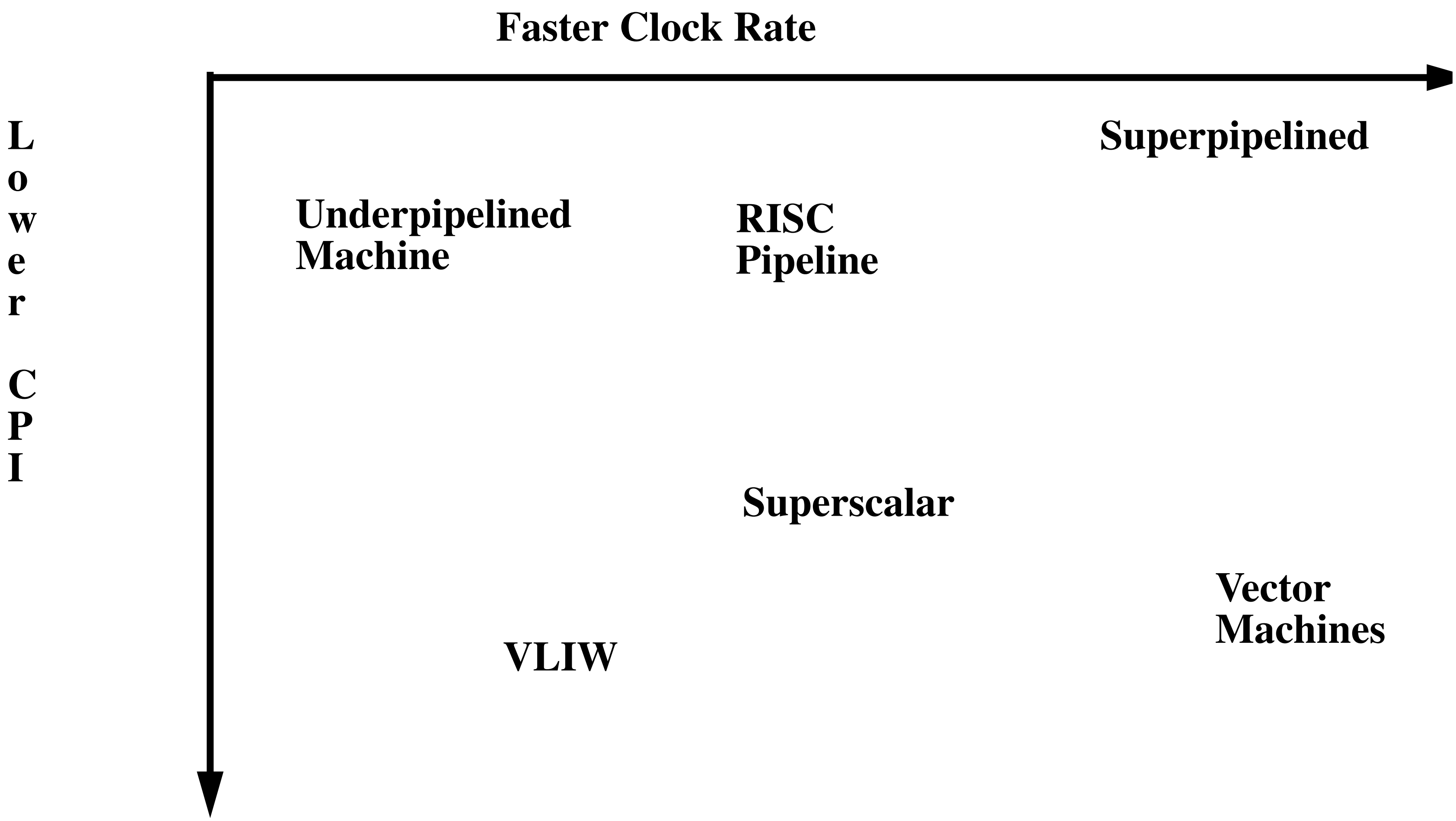
• Pipeline control unit must decide what to do next - often late in cycle.

• Frequently the bottleneck – especially in more complex machines.

• Simplified by simple, one-cycle RISC instructions – eliminates partially completed instructions.

• Less time constrained since all writes at the end of the cycle.

Architectural Approaches



Superscalar

Instruction		Pipeline Stages						
Type 1	IF	ID	EX	MEM	WB			
Type 2	IF	ID	EX	MEM	WB			
Type 1		IF	ID	EX	MEM	WB		
Type 2		IF	ID	EX	MEM	WB		
Type 1			IF	ID	EX	MEM	WB	
Type 2			IF	ID	EX	MEM	WB	
Type 1				IF	ID	EX	MEM	WB
Type 2				IF	ID	EX	MEM	WB

Superpipelined

Instruction	Pipeline Stages									
Type 1	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2
Type 2		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1
Type 1			IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2
Type 2				IF1	IF2	ID1	ID2	EX1	EX2	MEM1
Type 1					IF1	IF2	ID1	ID2	EX1	EX2
Type 2						IF1	IF2	ID1	ID2	EX1
Type 1							IF1	IF2	ID1	ID2
Type 2								IF1	IF2	ID1n

Superscalar vs. Superpipelined

- n-way superscalar and n-deep superpipelined have about equal performance (assuming the same issue restrictions).

- Superpipelined consumes fewer resources, but requires higher clock speeds.

- Actual performance differences are dictated by:
  - Maximum issue rate.
  - Issue restrictions: what instruction types can issue together. Most important factor.
  - Pipeline depths that affect total parallelism needed to keep the machine busy.

– Instruction alignment effects and branch effects.

What Limits Instruction Level Parallelism?

Application parallelism:

- **Limited parallelism in the program.**
- **Particularly serious problems for nonscientific code.**

Machine parallelism:

- **Issue limitations: number of issues and constraints**
- **Limited functional units**

Final Thoughts and Open Issues

- Techniques to extract further instruction-level parallelism + Better technology => performance growth of 1.5–2x per year for several more years.

- Much closer integration with technologists will be required in about 3 years with clock speeds > 100 MHz.

- Do RISC ideas run out of steam in the mid-1990s?

- Why? Limitations in our ability to extract more instruction level parallelism efficiently. Clock rates already high.

- If so, growth returns to technology-driven 20%-25%/year.

- Longer-term direction–Parallel Processing

- But watch out for growth in uniprocessors.

- Will be much harder to make commercially successful–may need long time to move technology to industry.