

Comparison of CISC and Early RISC Machines

	CISC			RISC		
	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berkeley RISC1	Stanford MIPS
year completed	1973	1978	1978	1980	1981	1983
instructions	208	303	270	120	39	55
microcode size (bytes)	54K	61K	17K	0	0	0
instruction size (bytes)	2-6	2-57	1-3	4	4	4
execution model	reg-reg reg-mem mem-mem	reg-reg reg-mem mem-mem	stack	reg-reg	reg-reg	reg-reg

Characteristics of RISC and CISC Machines

	RISC	CISC
1	fixed length instructions	variable length instructions
2	only loads/stores reference memory	any instruction may reference memory
3	few instructions and modes	many instructions and modes
4	instructions executed by the hardware	instructions interpreted by the microcode
5	simple instructions taking 1 cycle	complex instructions taking multiple cycles
6	highly pipelined	not or less pipelined
7	complexity is in the compiler	complexity is in the microprogram
8	multiple register sets	single register set

RISC is now the dominant processor architecture
(Pentium implements CISC using RISC!)

RISC Processors

First generation characteristics:

- uniform instruction length
- load/store architecture (simple addressing)
- pipelining
- branching (delayed branching and branch prediction)

Second generation:

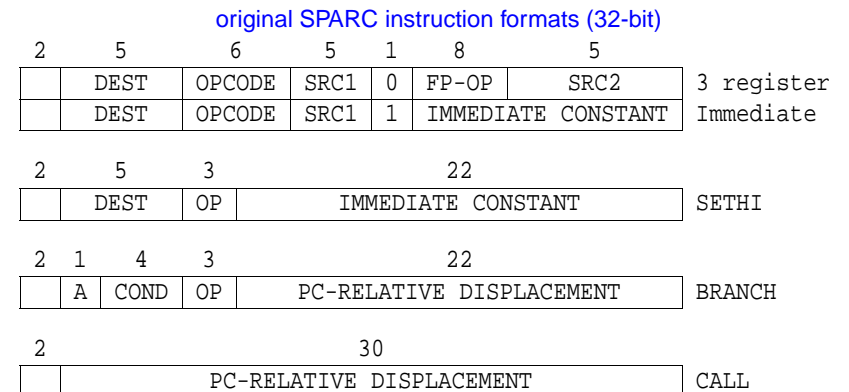
- faster clocks
- super-pipelining
- superscalar

Post-RISC

- out-of-order execution

Uniform Instruction Length

- fundamental RISC feature (and also true for PeANUt - 16 bit)
- CISC machines have apriori knowledge of length of each instruction



Load/Store Architecture

- memory reference restricted to load/store operations
 - NOT true for PeANUt, with 4 modes (direct, !, *, @) available to arithmetic instructions (e.g. `add @x`)
 - BUT we have many more registers for variables (not just AC)
- operations only occur between data that is in registers

```

! e.g. a = a + b
ld    [%o1], %i1    ! %i1 = memory [%o1]
ld    [%o2], %i2    ! %i2 = memory [%o2]
add   %i1, %i2, %i3 ! %i3 = %i1 + %i2
st    %i3, [%o1]    ! memory [%o1] = %i3
    
```

- motivation:
 - to enable fixed instruction length
 - to ease implementation of pipelining (see below)
 - to reduce the number of (slow) memory references

The Instruction Execution Cycle and Pipelining

- recall the PeANUt Instruction Execution Cycle (lect P1, p5) has a number of stages
 - one operation completes before the next starts; `dvd @x` is very slow!
- RISC needed fast cycle times, but cycle time reflects complexity of the operation
- RISC issues one instr'n per cycle, but does NOT require its completion in one cycle
- RISC can complete one instr'n per cycle via pipelining
 - break instr'n execution into k stages; \Rightarrow can get $\leq k$ -way parallelism

(generally, the circuitry for each stage is independent)

- e.g. ($k = 5$): stages FI = Fetch Instr'n., DI = Decode Instr'n., FO = Fetch Operand, EX =

Execute Instr'n., WB = Write Back

(branch):	FI	DI	FO	EX	WB				
(delay slot:)		FI	DI	FO	EX	WB			
(guess)			FI	DI	FO	EX	WB		
(guess)				FI	DI	FO	EX	WB	
(sure)					FI	DI	FO	EX	WB

- note: EX & WB stages may involve memory accesses (and may possibly stall the pipeline)

Branch Delay and Prediction

- branch to a new program address will disrupt the pipeline flow
 - processor doesn't know if instruction is a branch until the decode stage
 - may not know if branch taken until execute stage
 - if taken, "in flight" instructions must be annulled

- SPARC has a 'branch delay slot' instr'n immediately after any branch instructions
 - e.g.

```

cmp %i1, %i2    !
add %i3, 1, %i3 ! n = n + 1
bne endif1     ! if (i != k) ...
nop            ! /*branch delay slot; ALWAYS executed*/
    
```

(could move the add into the branch delay slot)

- pipeline can continue for *unconditional* branches (e.g. function calls)
- conditional branches are more difficult, pipeline will stall and require flushing as it can't be recognized before the DI stage
 - hardware for branch prediction is very important

Superscalar (2nd Generation Feature)

a small number (w) of instr'ns (a group) are scheduled by the H/W to *execute together*

- groups must have an appropriate 'instruction mix'
 - e.g. UltraSPARC ($w = 4$): $\left. \begin{array}{l} \leq 2 \text{ different floating point} \\ \leq 1 \text{ load / store ; } \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right\}$ instr'ns per group

- have $\leq w$ -way parallelism over *different* types of instr'ns

- generally requires:
 - multiple ($\geq w$) instr'n fetches
 - extra grouping stage (G) in the pipeline

- problem: amplifies all problems with pipelining
- issue: the instruction mix must be balanced for maximum performance!

note: recent discussion on multiply instruction;

how to get a SPARC to execute 1 f.p. multiply (and add!) per cycle

SPARC Assembler #1

- examine `sum.s` produced via `gcc -S sum.c`

```
.section ".text"
.align 4
.global sum
.type sum,#function
sum:                ! int sum(int n1, int n2) {
!#PROLOGUE# 0      ! /* here %sp = %fp */
save %sp, -120, %sp ! /* allocate stack frame *
!#PROLOGUE# 1      ! int tmp;
st %i0, [%fp+68]   ! /* save n1 on stack */
st %i1, [%fp+72]   ! /* save n2 */
ld [%fp+72], %o1   ! tmp = n1 + n2;
add %o0, %o1, %o0 !
st %o0, [%fp-20]  ! /* slot for tmp on stack
ld [%fp-20], %o0  ! return tmp;
mov %o0, %i0      ! /* %o0 holds RV */
b .LL1           ! /* a pointless jump! */
nop              ! /* delay slot */
.LL1:             !
ret              !
restore         ! } /* deallocate frame */
```

SPARC Assembler #2

- examine `sum.s` produced via `gcc -S -O sum.c` (and for `repeat.c`)

```
sum:                ! int sum(int n1, int n2) {
!#PROLOGUE# 0      ! int tmp; /* %o0=n1,&o1=n2 */
!#PROLOGUE# 1      ! tmp = n1 + n2;
retl               ! return tmp; /* 'leaf' function*/
add %o0, %o1, %o0 ! } /* %o0 holds RV*/
repeat:           ! int repeat(int n) {
!#PROLOGUE# 0      ! /* %o0=n */
!#PROLOGUE# 1      !
mov %o0, %g3      ! int tmp, i; /* %g3=n */
mov 0, %o0        ! tmp = 0; /* %o0=tmp */
cmp %o0, %g3      ! /* for --> if - do - while */
bge .LL4         !
mov %o0, %g2      ! for (i=0; i<n; i++) /* %g2=i */
add %o0, %g2, %o0 ! tmp = tmp + i;
.LL8:             !
add %g2, 1, %g2   !
cmp %g2, %g3      !
bl,a .LL8        !
add %o0, %g2, %o0 !
.LL4:             !
retl               ! return tmp; /* %o0 holds RV */
nop              ! }
```