Computer Organization and Architecture

Chapter 13

Reduced Instruction Set Computers (RISC)

Major Advances in Computers(1)

- The family concept
  - — IBM System/360 1964
  - — DEC PDP-8
  - — Separates architecture from implementation
- Microprogrammed control unit
  - — Idea by Wilkes 1951
  - — Produced by IBM S/360 1964
  - — Simplifies design and implementation of control unit
- Cache memory
  - — IBM S/360 model 85 1969

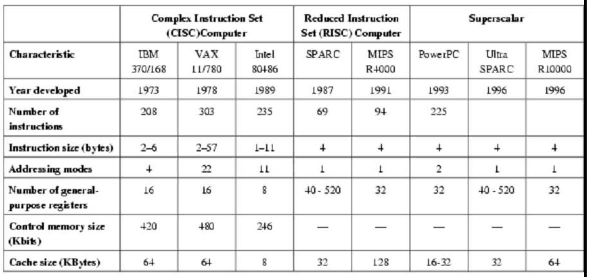
Major Advances in Computers(2)

- Solid State RAM
  - — (See memory notes)
- Microprocessors
  - — Intel 4004 1971
- Pipelining
  - — Introduces parallelism into fetch execute cycle
- Vector processing
  - — Explicit parallelism
- Multiple processors
- RISC design

RISC

- Reduced Instruction Set Computer
  - — A dramatic departure from historical architectures
- Key features
  - — Large number of general purpose registers
  - — or use of compiler technology to optimize register use
  - — Limited and simple instruction set
  - — Emphasis on optimizing the instruction pipeline

Comparison of Processors



	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Driving force for CISC

- Software costs far exceed hardware costs
- Increasingly complex high level languages
- Semantic gap: hard to translate from HLL semantics to machine semantics
- Leads to:
  - — Large instruction sets
  - — More addressing modes
  - — Hardware implementations of HLL statements
    - – e.g. CASE (switch) in VAX
    - – LOOP in Intel x86

Intent of CISC

- Ease compiler writing
- Improve execution efficiency
  - Complex operations in microcode
- Support more complex HLLs

Execution Characteristics

- Operations performed
  - Determine functions to be performed by processor and its interaction with memory
- Operands used
  - Determine instruction format and addressing modes
- Execution sequencing
  - Determines control and pipeline organization
- Studies have been done based on programs written in HLLs
- Dynamic studies are measured during the execution of the program

Operations

- Assignments
  - Movement of data
- Conditional statements (IF, LOOP)
  - Sequence control
- Procedure call-return is very time consuming
- Some HLL instructions lead to many machine code operations

Weighted Relative Dynamic Frequency of HLL Operations

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

- Weights
  - Machine instruction: Multiply cols 2 & 3 by number of machine instructions
  - Memory reference: Multiply cols 2 & 3 by number of memory references

Operands

- Mainly local scalar variables
- Optimisation should concentrate on accessing local variables

	Pascal	C	Average
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	55%
Array/Structure	26%	24%	25%

Procedure Calls

- Very time consuming operation
- Depends on number of parameters passed
  - Great majority use few parameters
  - 90% use three or fewer
- Procedure invocation depth
  - Fairly shallow for most programs
- Most variables are local and scalar
  - cache or registers

Implications

- Best support is given by optimising most used and most time consuming features
- Large number of registers
  - To optimize operand referencing
- Careful design of pipelines
  - Optimal branch handling important
- Simplified (reduced) instruction set

Operands and Registers

- Quick access to operands is desirable.
  - Many assignment statements
  - Significant number of operand accesses per HLL statement
- Register storage is fastest available storage
- Addresses are much shorter than memory or cache
- So we'd like to keep operands in registers as much as possible

Large Register File

- Software approach
  - Require compiler to allocate registers
  - Allocate based on most used variables in a given time
  - Requires sophisticated program analysis to allocate registers efficiently
  - Can be very difficult on architectures such x86 where registers have special purposes
- Hardware approach
  - Have more registers
  - Thus more variables will be in registers

Using Registers for Local Variables

- Store local scalar variables in registers to reduces memory access
- Every procedure (function) call changes locality
  - Save variables in registers
  - Pass Parameters
  - Get return results
  - Restore variables

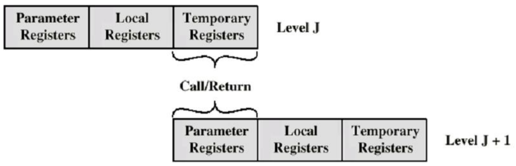
Register Windows

- Because most calls use only a few parameters and call depth is typically shallow:
  - Use multiple small sets of registers
  - Calls will switch to a different set of registers
  - Returns will switch back to a previously used set of registers

Register Windows cont.

- We can divide a register set into 3 areas:
  1. Parameter registers
  2. Local registers
  3. Temporary registers
- Temporary registers from one set overlap parameter registers from the next
- This allows parameter passing without actually moving or copying any data

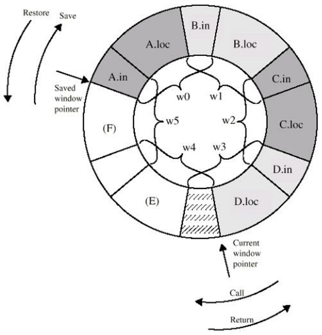
Overlapping Register Windows



Depth of Call Stack

- To handle any possible pattern of call and return, the number of register windows would have to be unbounded – clearly an impossibility
- When call depth exceeds the number of available register windows, older activations have to be saved in memory and restored later when call depth decreases
- A circular buffer organization can make this reasonably efficient

Circular Buffer



Operation of Circular Buffer

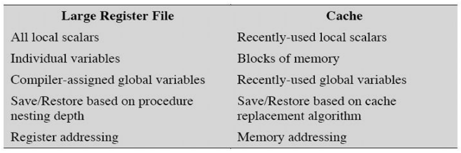
- When a call is made, a current window pointer is moved to show the currently active register window
- A saved window pointer indicates where the next saved window should restore
- When a CALL causes all windows to be in use (CWP is incremented and becomes equal to SWP), an interrupt is generated and the oldest window (the one furthest back in the call nesting) is saved to memory
- When a Return decrements CWP and it becomes equal to SWP an interrupt is generated and registers are restored from memory

Global Variables

- The register window scheme provides an efficient way to allocate registers to local variables but does not address global (static) variables
- The most straightforward solution is to allocate all globals to memory and never use registers to store than
  - Easy to implement
  - Inefficient for frequently accessed variables
- Or we can have a set of registers for global variables
  - Increases hardware and compiler complexity
  - Typically only a few globals can stored in regs

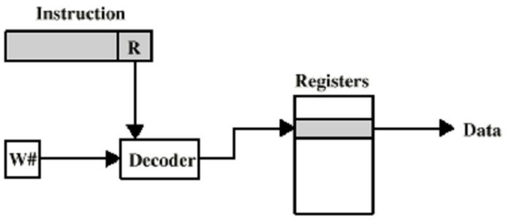
Registers vs. Cache

- Register file acts like a fast cache; would it be simpler and better to use a small register file and cache variables?

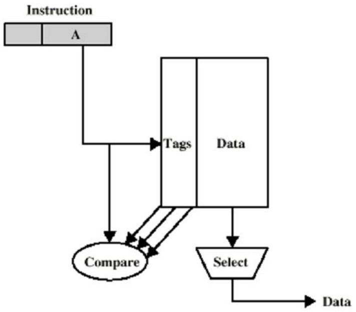


Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing

Referencing a Scalar - Window Based Register File



Referencing a Scalar - Cache



Registers vs. Cache

- No clear-cut choice
- But register files have simpler and therefore faster addressing
- When L1 (and possibly L2) cache are on-board cache memory access is almost as fast as register access
  - A rather confusing sentence from the text: “It should be clear that even if the cache is as fast as the register file the access time is will be considerably longer”
- See 13.6 (MIPS) for discussion of machine with large register file and cache

Compiler Based Register Optimization

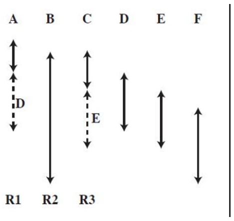
- Assume small number of registers (16-32) available
- Optimizing register use is up to compiler
- HLLs do not support explicit references to registers
  - Except for C – ex. register int i;
- Goal of optimizing compiler is to maximize register usage and minimize memory accesses

Basic Approach

- Assign a symbolic or virtual register to each candidate variable
- Map (unlimited) symbolic registers to real registers
- Symbolic registers with usage that does not overlap in time can share real registers
- If you run out of real registers some variables use memory
- One commonly used algorithm is the graph coloring algorithm

Graph coloring example

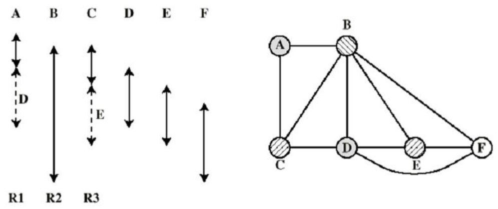
- We have six variables (symbolic registers) but only three actual registers available
- Analyze variable references over time to build a register interference graph



Graph Coloring

- Given a graph of nodes and edges:
  - Assign a color to each node
  - Adjacent nodes have different colors
  - Use minimum number of colors
- Nodes are symbolic registers
- Two registers that are live in the same program fragment are joined by an edge
- Try to color the graph with *n* colors, where *n* is the number of real registers
  - Nodes that are joined by an edge must have different colors
- Nodes that cannot be colored are allocated in memory

Graph Coloring Approach



(a) Time sequence of active use of registers

(b) Register interference graph

Why CISC (1)?

- Compiler simplification
  - But complex machine instructions are harder to exploit well
  - Machine code optimization is more difficult
- Smaller programs?
  - Program takes up less memory but…
  - Memory is now cheap
  - May not occupy less bits, just look shorter in symbolic form
    - More instructions require longer op-codes
    - Register references require fewer bits

Why CISC (2)?

- Faster programs?
  - There is a bias towards use of simpler instructions despite the efficiency of complex instructions
  - CISC machines need a more complex control unit and/or larger microprogram control store so simple instructions may take longer to execute
  - Note in Pentium that instruction cache is actually microcode storage
    - Microcode consists of micro-ops – RISC instructions that execute in the RISC core

RISC Characteristics

- One instruction per machine cycle
  - Cycle = time needed to fetch two operands from registers, perform ALU op, store result in register
  - Simple hardwired instructions need little or no microcode
- Register to register operations
  - Reduces variations in instruction set (e.g., VAX has 25 add instructions)
  - Memory access Load and Store only

RISC Characteristics

- Few, simple addressing modes
  - Complex addressing modes can by synthesized in software from simpler ones
- Few, simple instruction formats
  - Fixed length instruction format
  - Aligned on word boundaries
  - Fixed field locations especially the opcode
  - Simpler decode circuitry

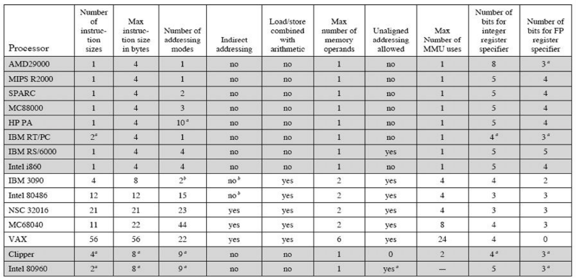
RISC vs CISC

- Not clear cut
  - Many studies fail to distinguish the effect of a large register file from the effect of RISC instruction set
- Many designs borrow from both philosophies
  - E.g. PowerPC and Pentium
  - RISC and CISC appear to be converging

Classic RISC Characteristics in Detail

1. A single instruction size, typically 4 bytes
2. Small number of addressing modes
3. No memory-indirect addressing
4. No operations combine load/store with arithmetic
5. No more than one memory addressed operand per instruction
6. Does not support arbitrary (byte) alignment of data for load/store
7. Max number of MMU uses for a data address is 1
8. At least 5 bits for integer register specifier (32 registers)
9. At least 4 bits for FP register specifier (16 registers)

Processor Characteristics



Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[b]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

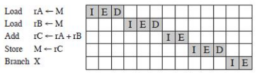
a RISC that does not conform to this characteristic.
b CISC that does not conform to this characteristic.

RISC Pipelining

- Most instructions are register to register
- Two phases of execution
  - I: Instruction fetch
  - E: Execute
    - ALU operation with register input and output
- For load and store
  - I: Instruction fetch
  - E: Execute
    - Calculate memory address
  - D: Memory
    - Register to memory or memory to register operation
- Execute can be further subdivided:
  - E1: Register file read
  - E2: ALU operation and register write

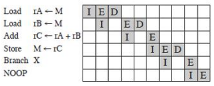
Sequential Execution & 2-Stage Pipeline

- In the 2 stage pipeline I and E (Fetch and Execute) can be performed in parallel but not D (reg/mem operation)
  - Single port memory allows only one access per stage
  - Insert a WAIT stage where D operations occur
  - Use a No-op (NOOP or NOP) to keep the pipeline full when branch executes (minimizes circuitry needed to handle pipeline stall)



Load	$rA \leftarrow M$	I	E	D										
Load	$rB \leftarrow M$				I	E	D							
Add	$rC \leftarrow rA + rB$							I	E					
Store	$M \leftarrow rC$									I	E	D		
Branch	X												I	E

(a) Sequential execution

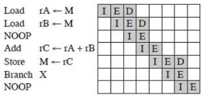


Load	$rA \leftarrow M$	I	E	D							
Load	$rB \leftarrow M$		I		E	D					
Add	$rC \leftarrow rA + rB$				I		E				
Store	$M \leftarrow rC$						I	E	D		
Branch	X							I		E	
NOOP										I	E

(b) Two-stage pipelined timing

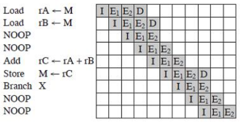
3 and 4 stage Pipelines

- Permitting 2 memory accesses per stage allows 3-stage pipeline with almost 3x speedup
- Divide E phase into two smaller phases for more even timing in 4 stage pipeline
- Use NOPs for pipeline delays (e.g., data dependencies)



Load	$rA \leftarrow M$	I	E	D					
Load	$rB \leftarrow M$		I	E	D				
NOOP				I	E				
Add	$rC \leftarrow rA + rB$				I	E			
Store	$M \leftarrow rC$					I	E	D	
Branch	X						I	E	
NOOP								I	E

(c) Three-stage pipelined timing



Load	$rA \leftarrow M$	I	E_1	E_2	D							
Load	$rB \leftarrow M$		I	E_1	E_2	D						
NOOP				I	E_1	E_2						
NOOP					I	E_1	E_2					
Add	$rC \leftarrow rA + rB$					I	E_1	E_2				
Store	$M \leftarrow rC$						I	E_1	E_2	D		
Branch	X							I	E_1	E_2		
NOOP									I	E_1	E_2	
NOOP										I	E_1	E_2

(d) Four-stage pipelined timing

Optimization of Pipelining

- Use NOPs (No Operation) instead of hardware
  - Compiler inserts NOPs when data dependencies or branches appear
- Delayed branch
  - A Branch does not take effect until after execution of following instruction
  - This following instruction is the delay slot

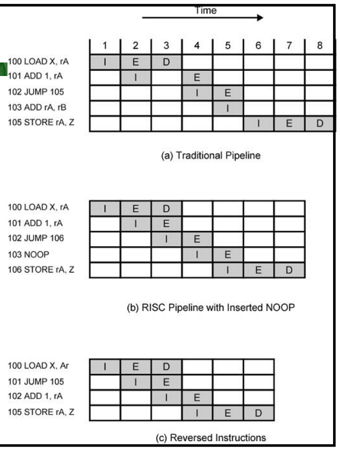
Rationale for Delayed Branch

- A branch's E stage occurs after the fetch of the next instruction
  - For unconditional branches the fetch is wasted
  - For conditional branches it is risky
- So switch order of branch and its preceding instruction B_{-1}:
  - Fetch branch
  - Fetch B_{-1} – the branch does not update PC until execution is complete
  - Then execute B_{-1} while fetching target of branch

Normal and Delayed Branch

Address	Normal Branch	Delayed Branch	Optimized Delayed Branch
100	LOAD X, rA	LOAD X, rA	LOAD X, rA
101	ADD 1, rA	ADD 1, rA	JUMP 105
102	JUMP 105	JUMP 106	ADD 1, rA
103	ADD rA, rB	NOOP	ADD rA, rB
104	SUB rC, rB	ADD rA, rB	SUB rC, rB
105	STORE rA, Z	SUB rC, rB	STORE rA, Z
106		STORE rA, Z	

Use of Delayed Branch



	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

(a) Traditional Pipeline

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC Pipeline with Inserted NOOP

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed Instructions

When can delayed branch be used

- Always OK for
  - Unconditional branches
  - Calls
  - Returns
- With conditional branches if the condition can be altered by the branch delay candidate instruction (the one immediately preceding the branch) then a NOP has to be used

Problems with delay slots

- General idea is to find an instruction that is safe to execute regardless of which way the branch goes
  - Compiler or assembler is responsible
  - Increases compiler/assembler complexity
  - Could lead to non-portable programs that have different results if processor does not have a delay slot
  - Problem for systems programmers

Delayed Load

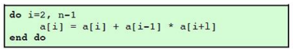
- When a load from memory is encountered processor locks target register
- Then continues executing instruction stream until it reaches an instruction that requires the load target

Loop Unrolling

- A compiler technique for improving instruction parallelism
- Repeat body of a loop in code some number of times *u* (the unrolling factor) and iterate by step *u* instead of step 1
- Improves performance by
  - Reduce loop overhead (branching)
  - Increase parallelism by improving pipeline performance
  - Improve register, cache and/or TLB locality

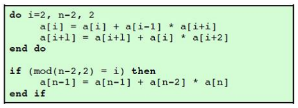
Loop unrolling example

- 2nd assignment performed while first is being stored and loop variable updated
- If array elements are in regs then locality of reference will improve because a[I] and a[I+1] are used twice, reducing loads per iteration from 3 to 2



```
do i=2, n-1
     a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) original loop



```
do i=2, n-2, 2
     a[i] = a[i] + a[i-1] * a[i+i]
     a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2,2) = i) then
     a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) loop unrolled twice

MIPS R4000

- MIPS Technology Inc developed one of the first commercially available RISC processors
- R4000 is a 64-bit machine
  - Used for all buses, data paths, registers and addresses
- Major processor units are the CPU and a coprocessor for memory management
- Processor has a very simple architecture
  - 32 64 bit registers
  - 64K instruction and data caches (R3000)
  - Register $0 is a constant 0
  - Register $31 is a link register

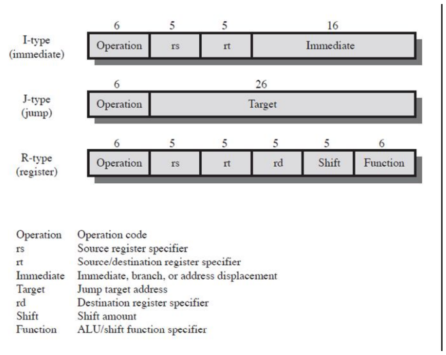
Scicortex Supercomputer

- Based on MIPS 4000 specification
- Each multicore node has:
  - 6 MIPS cores
  - Crossbar memory controller
  - Interconnect DMA engine
  - Gigabit ethernet
  - PCI express controller
- In a single chip that consumes 10 watts of power and is capable of 6 GFLOPs (floating point operations per second)

Instruction Set Design

- All instructions are 32 bits
- All data operations are reg/reg
- Only memory operations are load and store
- No condition code register
  - Conditional instructions generate flags in a GP register
  - No need for special CC handling in a pipeline
  - Registers subject to same dataflow analysis as normal operands
  - Conditions mapped to GP registers are also subject to compile-time register analysis

Instruction Formats



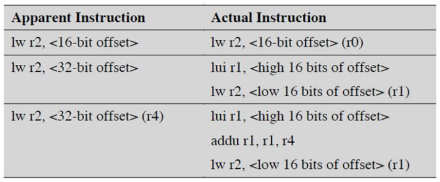
Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier

Memory Addressing

- Textbook actually discusses the MIPS R3000 instruction set
- Memory references are 16-bit offsets from a 32-bit register

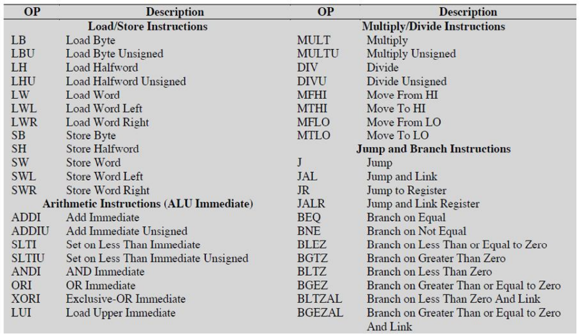
Synthesizing Other addressing modes

- Note limitations on address length imposed by the 32-bit instruction format



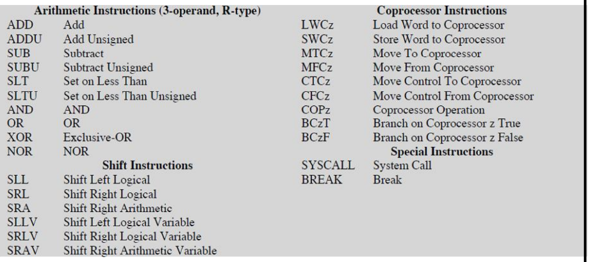
Apparent Instruction	Actual Instruction
lw r2, <16-bit offset>	lw r2, <16-bit offset> (r0)
lw r2, <32-bit offset>	lui r1, <high 16 bits of offset> lw r2, <low 16 bits of offset> (r1)
lw r2, <32-bit offset> (r4)	lui r1, <high 16 bits of offset> addu r1, r1, r4 lw r2, <low 16 bits of offset> (r1)

Instruction Set (1)



OP	Description	OP	Description
Load/Store Instructions		**Multiply/Divide Instructions**	
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word	MFHI	Move From HI
LWL	Load Word Left	MTHI	Move To HI
LWR	Load Word Right	MFLO	Move From LO
SB	Store Byte	MTLO	Move To LO
SH	Store Halfword	**Jump and Branch Instructions**	
SW	Store Word	J	Jump
SWL	Store Word Left	JAL	Jump and Link
SWR	Store Word Right	JR	Jump to Register
Arithmetic Instructions (ALU Immediate)		JALR	Jump and Link Register
ADDI	Add Immediate	BEQ	Branch on Equal
ADDIU	Add Immediate Unsigned	BNE	Branch on Not Equal
SLTI	Set on Less Than Immediate	BLEZ	Branch on Less Than or Equal to Zero
SLTIU	Set on Less Than Immediate Unsigned	BGTZ	Branch on Greater Than Zero
ANDI	AND Immediate	BLTZ	Branch on Less Than Zero
ORI	OR Immediate	BGEZ	Branch on Greater Than or Equal to Zero
XORI	Exclusive-OR Immediate	BLTZAL	Branch on Less Than Zero And Link
LUI	Load Upper Immediate	BGEZAL	Branch on Greater Than or Equal to Zero And Link

Instruction Set (2)



Arithmetic Instructions (3-operand, R-type)		**Coprocessor Instructions**	
ADD	Add	LWCz	Load Word to Coprocessor
ADDU	Add Unsigned	SWCz	Store Word to Coprocessor
SUB	Subtract	MTCz	Move To Coprocessor
SUBU	Subtract Unsigned	MFCz	Move From Coprocessor
SLT	Set on Less Than	CTCz	Move Control To Coprocessor
SLTU	Set on Less Than Unsigned	CFCz	Move Control From Coprocessor
AND	AND	COPz	Coprocessor Operation
OR	OR	BCzT	Branch on Coprocessor z True
XOR	Exclusive-OR	BCzF	Branch on Coprocessor z False
NOR	NOR	**Special Instructions**	
Shift Instructions		SYSCALL	System Call
SLL	Shift Left Logical	BREAK	Break
SRL	Shift Right Logical		
SRA	Shift Right Arithmetic		
SLLV	Shift Left Logical Variable		
SRLV	Shift Right Logical Variable		
SRAV	Shift Right Arithmetic Variable		

MIPS Pipeline

- First generation RISC processors achieved throughput of almost 1 instruction/clock
- Two classes of processors improve on this:
  - — Superscalar (2 or more separate pipelines)
  - — Superpipelined (more stages in pipeline)
- R4000 is a super-pipelined architecture
- Changes from R3000 to R4000 exemplify modern pipeline technology

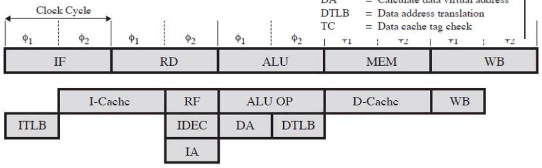
R3000 Pipeline

- Advances once per clock cycle
- MIPS compilers can fill delay slot ~80% of the the time
- Five stages:
  - Instruction fetch
  - Source operand fetch from register file
  - ALU operation or data operand address generation
  - Data memory reference
  - Write back into register file
- 60 ns clock divided into two 30ns stages

R3000 Pipeline

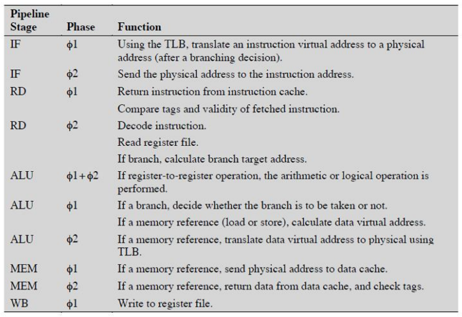
- Some parallelism within execution of a single instruction

IF = Instruction fetch
RD = Read
MEM = Memory access
WB = Write back
I-Cache = Instruction cache access
RF = Fetch operand from register
D-Cache = Data cache access
ITLB = Instruction address translation
IDEC = Instruction decode
IA = Compute instruction address
DA = Calculate data virtual address
DTLB = Data address translation
TC = Data cache tag check



(a) Detailed R3000 pipeline

Pipeline stage detail

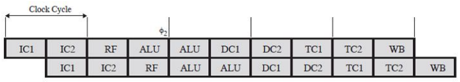


Pipeline Stage	Phase	Function
IF	ϕ1	Using the TLB, translate an instruction virtual address to a physical address (after a branching decision).
IF	ϕ2	Send the physical address to the instruction address.
RD	ϕ1	Return instruction from instruction cache. Compare tags and validity of fetched instruction.
RD	ϕ2	Decode instruction. Read register file. If branch, calculate branch target address.
ALU	ϕ1 + ϕ2	If register-to-register operation, the arithmetic or logical operation is performed.
ALU	ϕ1	If a branch, decide whether the branch is to be taken or not. If a memory reference (load or store), calculate data virtual address.
ALU	ϕ2	If a memory reference, translate data virtual address to physical using TLB.
MEM	ϕ1	If a memory reference, send physical address to data cache.
MEM	ϕ2	If a memory reference, return data from data cache, and check tags.
WB	ϕ1	Write to register file.

R4000 changes

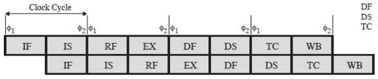
- On-chip cache is indexed by virtual addr rather than physical address
  - Cache lookup and address translation can be performed in parallel
- Use pipeline several times per clock cycle by dividing into smaller stages
  - Clock rate is multiplied by 2 or more
- R4000 had faster adder allowing ALU ops to proceed at double the R3000 rate
- Loads and stores were optimized to double the rate

R3000 Theoretical and actual R4000 superpipelines



(a) Superpipelined implmentation of the optimized R3000 pipeline

IF = Instruction fetch first half
IS = Instruction fetch second half
RF = Fetch operands from register
EX = Instruction execute
IC = Instruction cache
DC = Data cache
DF = Data cache first half
DS = Data cache second half
TC = Tag check



(b) R4000 pipeline

R4000 Pipeline stages

- 8 stages
  - Instruction Fetch 1st half
    - Virtual addr presented to instruction cache and TLB
  - Instruction Fetch 2nd half
    - I cache outputs instruction and TLB generates physical address
  - Register File (3 activities in parallel)
    - Instruction decode and check for data dependencies
    - Instruction cache tag check
    - Operands are fetched from register file
  - Instruction execute (one of 3 activities):
    - For reg/reg ALU performs the instruction
    - For load / store calculate data virtual address
    - For branch, calculate target virtual address and check branch conditions

R4000 Pipeline stages (2)

- Data cache 1st stage
  - Virtual address presented to data cache and TLB
- Data cache 2nd stage
  - Data cache outputs the instruction and TLB generates the physical address
- Tag check
  - Cache tag checks performed for loads and stores
- Write back
  - Instruction result is written back to the register file

SPARC

- Scalable Processor Architecture designed by Sun Microsystems
  - Implemented by Sun and licensed to other vendors
  - Inspired by Berkeley RISC I

SPARC Registers

- Uses register windows, each with 24 registers
- Number of windows ranges from 2 to 32
- Each procedure has a set of 32 logical registers 0 through 31
  - Physical registers 0-7 are global and shared by all processes / procedures
  - Logical registers 24-31 are “ins”; shared with with calling (parent) procedure
  - Logical registers 8-15 are “outs”; shared with called (child) procedure
  - These portions overlap in register windows
  - Logical registers 16-23 are locals and do not overlap

Procedure calls

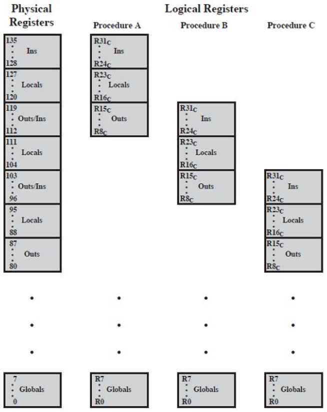
- Calling procedures load parameters in the *outs* registers
- Called procedure sees these as the *ins*
- Processor maintains window state in the processor status register:
  - CWP current window pointer
  - WIM windows invalid mask
- Saving and restoring registers for procedure calls is not normally necessary
- Compiler can be concernd primarily with local optimization

SPARC Instruction Set

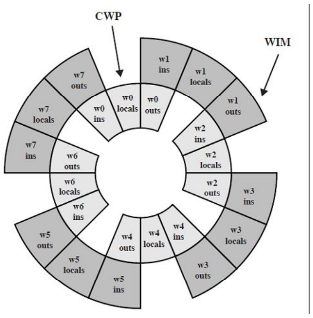
- Most instructions have only register operands
- Three-address format:

  Dest <- Source1 op Source2
- Dest and Source1 are registers
- Source2 is either a register or a 13-bit immediate operand
- Register R0 is hardwired 0

Register Layout with three procedures



Another view of register layout



Instruction Set (1)

OP	Description	OP	Description
	Load/Store Instructions		**Arithmetic Instructions**
LDSB	Load signed byte	ADD	Add
LDSH	Load signed halfword	ADDCC	Add, set icc
LDUB	Load unsigned byte	ADDX	Add with carry
LDUH	Load unsigned halfword	ADDXCC	Add with carry, set icc
LD	Load word	SUB	Subtract
LDD	Load doubleword	SUBCC	Subtract, set icc
STB	Store byte	SUBX	Subtract with carry
STH	Store halfword	SUBXCC	Subtract with carry, set icc
STD	Store word	MULSCC	Multiply step, set icc
STDD	Store doubleword		**Jump/Branch Instructions**

Instruction Set (2)

STDD	Store doubleword		**Jump/Branch Instructions**
	Shift Instructions	BCC	Branch on condition
SLL	Shift left logical	FBCC	Branch on floating-point condition
SRL	Shift right logical	CBCC	Branch on coprocessor condition
SRA	Shift right arithmetic	CALL	Call procedure
	Boolean Instructions	JMPL	Jump and link
AND	AND	TCC	Trap on condition
ANDCC	AND, set icc	SAVE	Advance register window
ANDN	NAND	RESTORE	Move windows backward
ANDNCC	NAND, set icc	RETT	Return from trap
OR	OR		**Miscellaneous Instructions**
ORCC	OR, set icc	SETHI	Set high 22 bits
ORN	NOR	UNIMP	Unimplemented instruction (trap)
ORNCC	NOR, set icc	RD	Read a special register
XOR	XOR	WR	Write a special register
XORCC	XOR, set icc	IFLUSH	Instruction cache flush
XNOR	Exclusive NOR		
XNORCC	Exclusive NOR, set icc		

Arithmetic and Logical Instructions

- Note the lack of a divide instruction
- All arithmetic and logical instructions have condition code (CC) forms
  - Unlike MIPS, this machine has a condition code register
- Note the rich set of Booleans

Memory References

- Only load and store instructions reference memory
  - Note the load signed/unsigned to extend 8 or 16 bits to 32 bits
- One addressing mode:

  EA = R1 + S2

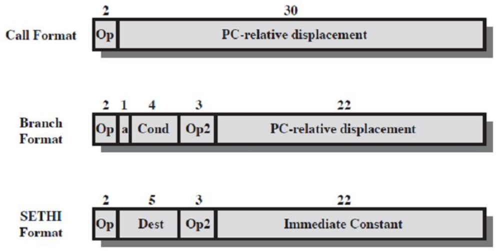
  Where R1 is a register and S2 is either an immediate displacement or a displacement in another register

Mode	Algorithm	SPARC Equivalent	Instruction Type
Immediate	operand = A	S2	Register-to-register
Direct	EA = A	R_0 + S2	Load, store
Register	EA = R	R_{S1}, R_{S2}	Register-to-register
Register Indirect	EA = (R)	R_{S1} + 0	Load, store
Displacement	EA = (R) + A	R_{S1} + S2	Load, store

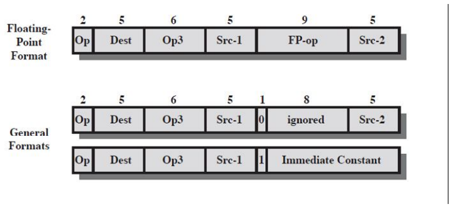
Instruction Format

- Fixed 32 bits
- Starts with 2-bit opcode (usually extended in other fields)
- Call instruction has 30 bits of address that are zero-extended on the right to form a 32-bit signed displacement
- Branch instructions can test any combination of condition codes
  - The “a” (annul) when clear cause the next instruction to be executed whether or not the branch is taken
- SETHI instruction is a special format to load and store 32-bit addresses and data

SPARC Instruction Formats (1)



SPARC Instruction Formats (2)



RISC/CISC Controversy

- Quantitative assessments
  - compare program sizes and execution speeds
- Qualitative assessments
  - examine issues of high level language support and use of VLSI real estate
- Problems
  - No pair of RISC and CISC processors exists that are directly comparable
  - No definitive set of test programs
  - Difficult to separate hardware effects from compiler effects
  - Most comparisons done on “toy” rather than production machines
  - Most commercial devices are a mixture

Convergence of Technologies

- RISC and CISC are converging
- RISC systems are becoming more complex
- CISC systems have focused on RISC techniques such as large number of registers and pipeline design