ARCHITECTURE DESIGN PRINCIPLES - IN 70's -

- Large microprograms would add little or nothing to the cost of the machine
  - ← Rapid growth of memory technology
  - ⇒ Large General Purpose Instruction Set
- Microprogram is much faster than the machine instructions
  - ← Microprogram memory is much faster than main memory
  - ⇒ Moving the software functions into microprogram for the high performance machines
- Execution speed is proportional to the program size
  - ← Architectural techniques that led to small program
  - ⇒ High performance instruction set
- Number of registers in CPU has limitations
  - ← Very costly
  - ⇒ Difficult to utilize them efficiently

COMPARISONS OF EXECUTION MODELS

A ← B + C Data: 32-bit

- **Register-to-register**

8	4	16	
Load	rB	B	
Load	rC	C	
Add	rA	rB	rC
Store	rA	A	

I = 104b; D = 96b; M = 200b

- **Memory-to-register**

8	16
Load	B
Add	C
Store	A

I = 72b; D = 96b; M = 168b

- **Memory-to-memory**

8	16	16	16
Add	B	C	A

I = 56b; D = 96b; M = 152b

FOUR MODERN ARCHITECTURES IN 70's

	IBM 370/168	DEC VAX-11/780	Xerox Dorado	Intel iAPX-432
Year	1973	1978	1978	1982
# of instrs.	208	303	270	222
Control mem. size	420 Kb	480 Kb	136 Kb	420 Kb
Instr. size (bits)	16-48	16-456	8-24	6-321
Technology	ECL MSI	TTL MSI	ECL MSI	NMOS VLSI
Execution model	reg-mem mem-mem reg-reg	reg-mem mem-mem reg-reg	stack	stack mem-mem
Cache size	64 Kb	64 Kb	64 Kb	64 Kb

COMPLEX INSTRUCTION SET COMPUTER

- These computers with many instructions and addressing modes came to be known as Complex Instruction Set Computers (CISC)

- One goal for CISC machines was to have a machine language instruction to match each high-level language statement type

VARIABLE LENGTH INSTRUCTIONS

- The large number of instructions and addressing modes led CISC machines to have variable length instruction formats
- The large number of instructions means a greater number of bits to specify them
- In order to manage this large number of opcodes efficiently, they were encoded with different lengths:
  - More frequently used instructions were encoded using short opcodes.
  - Less frequently used ones were assigned longer opcodes.
- Also, multiple operand instructions could specify different addressing modes for each operand
  - For example,
    - Operand 1 could be a directly addressed register,
    - Operand 2 could be an indirectly addressed memory location,
    - Operand 3 (the destination) could be an indirectly addressed register.
- All of this led to the need to have different length instructions in different situations, depending on the opcode and operands used

VARIABLE LENGTH INSTRUCTIONS

- For example, an instruction that only specifies register operands may only be two bytes in length
  - One byte to specify the instruction and addressing mode
  - One byte to specify the source and destination registers.
- An instruction that specifies memory addresses for operands may need five bytes
  - One byte to specify the instruction and addressing mode
  - Two bytes to specify each memory address
    - » Maybe more if there's a large amount of memory.
- Variable length instructions greatly complicate the fetch and decode problem for a processor
- The circuitry to recognize the various instructions and to properly fetch the required number of bytes for operands is very complex

COMPLEX INSTRUCTION SET COMPUTER

- **Another characteristic of CISC computers is that they have instructions that act directly on memory addresses**
  - **For example,**

    ADD L1, L2, L3

    that takes the contents of *M*[*L1*] adds it to the contents of *M*[*L2*] and stores the result in location *M*[*L3*]
- **An instruction like this takes three memory access cycles to execute**
- **That makes for a potentially very long instruction execution cycle**
- **The problems with CISC computers are**
  - **The complexity of the design may slow down the processor,**
  - **The complexity of the design may result in costly errors in the processor design and implementation,**
  - **Many of the instructions and addressing modes are used rarely, if ever**

SUMMARY: CRITICISMS ON CISC

High Performance General Purpose Instructions

- Complex Instruction
  - → Format, Length, Addressing Modes
  - → Complicated instruction cycle control due to the complex decoding HW and decoding process

- Multiple memory cycle instructions
  - → Operations on memory data
  - → Multiple memory accesses/instruction

- Microprogrammed control is necessity
  - → Microprogram control storage takes substantial portion of CPU chip area
  - → Semantic Gap is large between machine instruction and microinstruction

- General purpose instruction set includes all the features required by individually different applications
  - → When any one application is running, all the features required by the other applications are extra burden to the application

REDUCED INSTRUCTION SET COMPUTERS

- In the late ‘70s and early ‘80s there was a reaction to the shortcomings of the CISC style of processors
- Reduced Instruction Set Computers (RISC) were proposed as an alternative
- The underlying idea behind RISC processors is to simplify the instruction set and reduce instruction execution time

- RISC processors often feature:
  - Few instructions
  - Few addressing modes
  - Only load and store instructions access memory
  - All other operations are done using on-processor registers
  - Fixed length instructions
  - Single cycle execution of instructions
  - The control unit is hardwired, not microprogrammed

REDUCED INSTRUCTION SET COMPUTERS

- Since all but the load and store instructions use only registers for operands, only a few addressing modes are needed
- By having all instructions the same length, reading them in is easy and fast
- The fetch and decode stages are simple, looking much more like Mano's Basic Computer than a CISC machine
- The instruction and address formats are designed to be easy to decode
- Unlike the variable length CISC instructions, the opcode and register fields of RISC instructions can be decoded simultaneously
- The control logic of a RISC processor is designed to be simple and fast
- The control logic is simple because of the small number of instructions and the simple addressing modes
- The control logic is hardwired, rather than microprogrammed, because hardwired control is faster

ARCHITECTURAL METRIC

$A \leftarrow B + C$
$B \leftarrow A + C$
$D \leftarrow D - B$

- **Register-to-register (Reuse of operands)**

8	4	16	
Load	rB	B	
Load	rC	C	
Add	rA	rB	rC
Store	rA	A	
Add	rB	rA	rC
Store	rB	B	
Load	rD	D	
Sub	rD	rD	rB
Store	rD	D	

I = 228b
D = 192b
M = 420b

- **Register-to-register (Compiler allocates operands in registers)**

8	4	4	4
Add	rA	rB	rC
Add	rB	rA	rC
Sub	rD	rD	rB

I = 60b
D = 0b
M = 60b

- **Memory-to-memory**

8	16	16	16
Add	B	C	A
Add	A	C	B
Sub	B	D	D

I = 168b
D = 288b
M = 456b

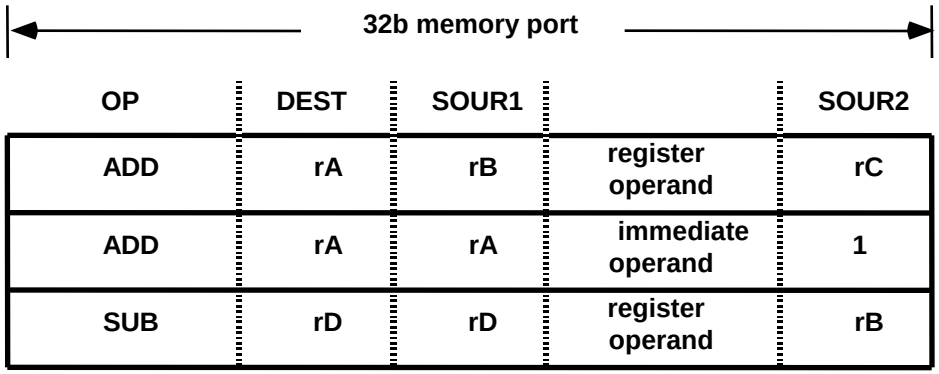
CHARACTERISTICS OF INITIAL RISC MACHINES

	IBM 801	RISC I	MIPS
Year	1980	1982	1983
Number of instructions	120	39	55
Control memory size	0	0	0
Instruction size (bits)	32	32	32
Technology	ECL MSI	NMOS VLSI	NMOS VLSI
Execution model	reg-reg	reg-reg	reg-reg

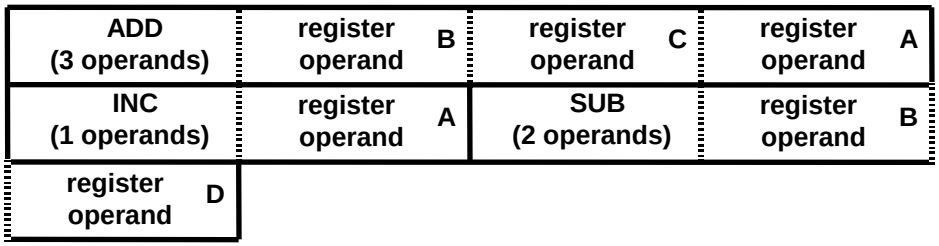
COMPARISON OF INSTRUCTION SEQUENCE

$A \leftarrow B + C$
$A \leftarrow A + 1$
$D \leftarrow D - B$

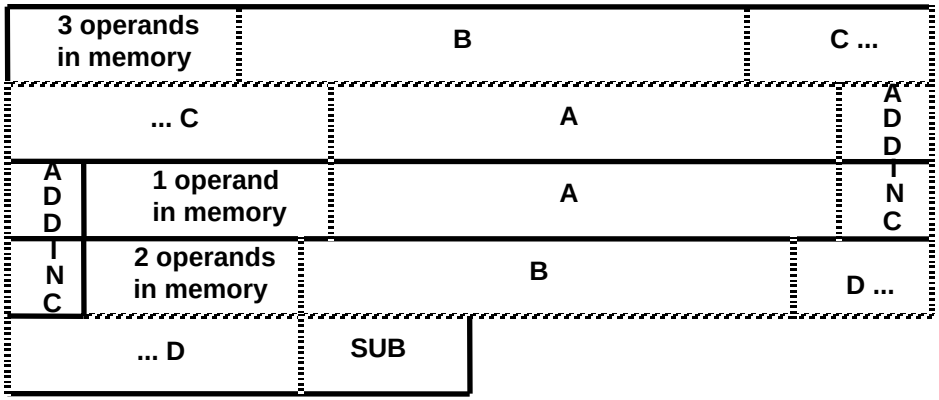
RISC 1



VAX



432



REGISTERS

- By simplifying the instructions and addressing modes, there is space available on the chip or board of a RISC CPU for more circuits than with a CISC processor
- This extra capacity is used to
  - Pipeline instruction execution to speed up instruction execution
  - Add a large number of registers to the CPU

PIPELINING

- A very important feature of many RISC processors is the ability to execute an instruction each clock cycle
- This may seem nonsensical, since it takes at least once clock cycle each to fetch, decode and execute an instruction.
- It is however possible, because of a technique known as pipelining
  - We'll study this in detail later
- Pipelining is the use of the processor to work on different phases of multiple instructions in parallel

PIPELINING

- For instance, at one time, a pipelined processor may be
  - Executing instruction i_t
  - Decoding instruction i_{t+1}
  - Fetching instruction i_{t+2} from memory
- So, if we're running three instructions at once, and it takes an average instruction three cycles to run, the CPU is executing an average of an instruction a clock cycle
- As we'll see when we cover it in depth, there are complications
  - For example, what happens to the pipeline when the processor branches
- However, pipelined execution is an integral part of all modern processors, and plays an important role

REGISTERS

- By having a large number of general purpose registers, a processor can minimize the number of times it needs to access memory to load or store a value
- This results in a significant speed up, since memory accesses are *much* slower than register accesses
- Register accesses are fast, since they just use the bus on the CPU itself, and any transfer can be done in one clock cycle
- To go off-processor to memory requires using the much slower memory (or system) bus
- It may take many clock cycles to read or write to memory across the memory bus
  - The memory bus hardware is usually slower than the processor
  - There may even be competition for access to the memory bus by other devices in the computer (*e.g.* disk drives)
- So, for this reason alone, a RISC processor may have an advantage over a comparable CISC processor, since it only needs to access memory
  - for its instructions, and
  - occasionally to load or store a memory value

UTILIZING RISC REGISTERS – REGISTER WINDOW

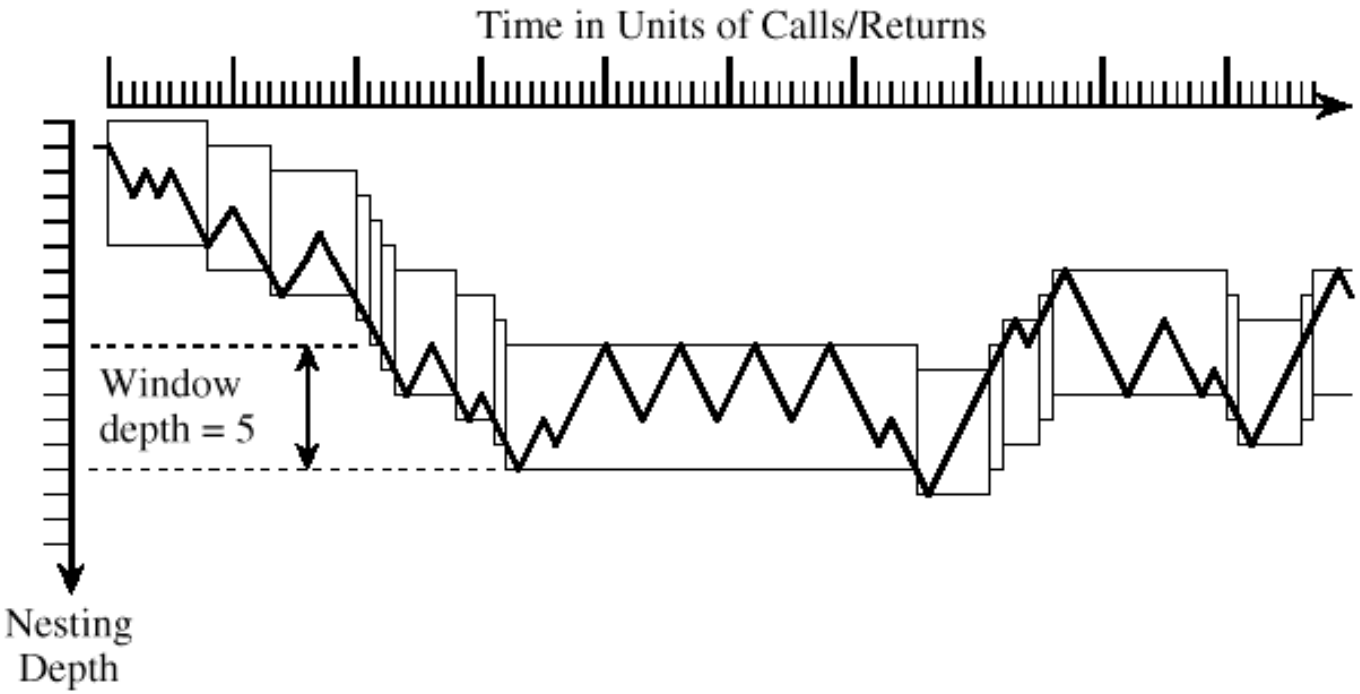
<Weighted Relative Dynamic Frequency of HLL Operations>

	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

⇒ **The procedure (function) call/return is the most time-consuming operations in typical HLL programs**

CALL-RETURN BEHAVIOR

Call-return behavior as a function of nesting depth and time



REGISTER WINDOW APPROACH

- **Observations**

  - **Weighted Dynamic Frequency of HLL Operations**
    ⇒ Procedure call/return is the most time consuming operations

  - **Locality of Procedure Nesting**
    ⇒ The depth of procedure activation fluctuates within a relatively narrow range

  - **A typical procedure employs only a few passed parameters and local variables**

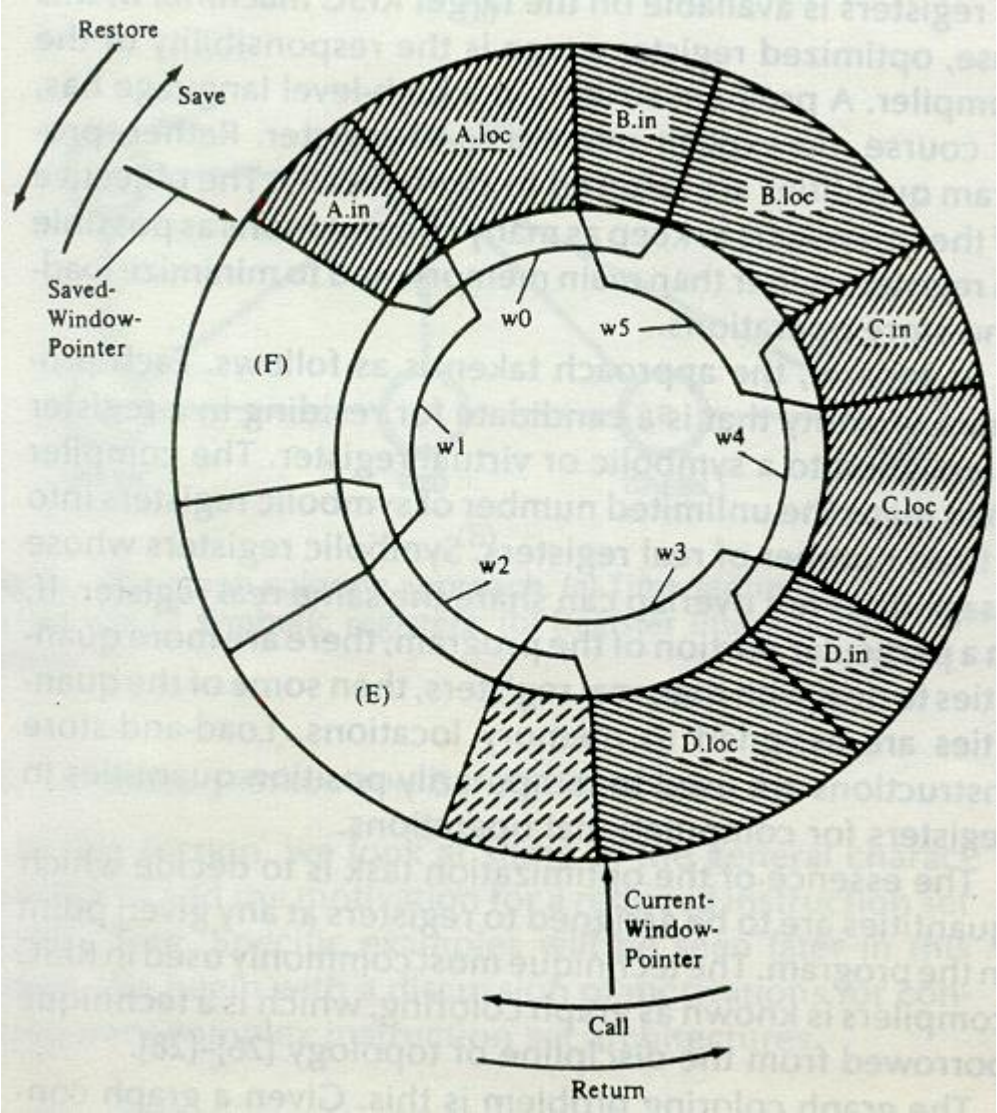
- **Solution**

  - **Use multiple small sets of registers (windows), each assigned to a different procedure**

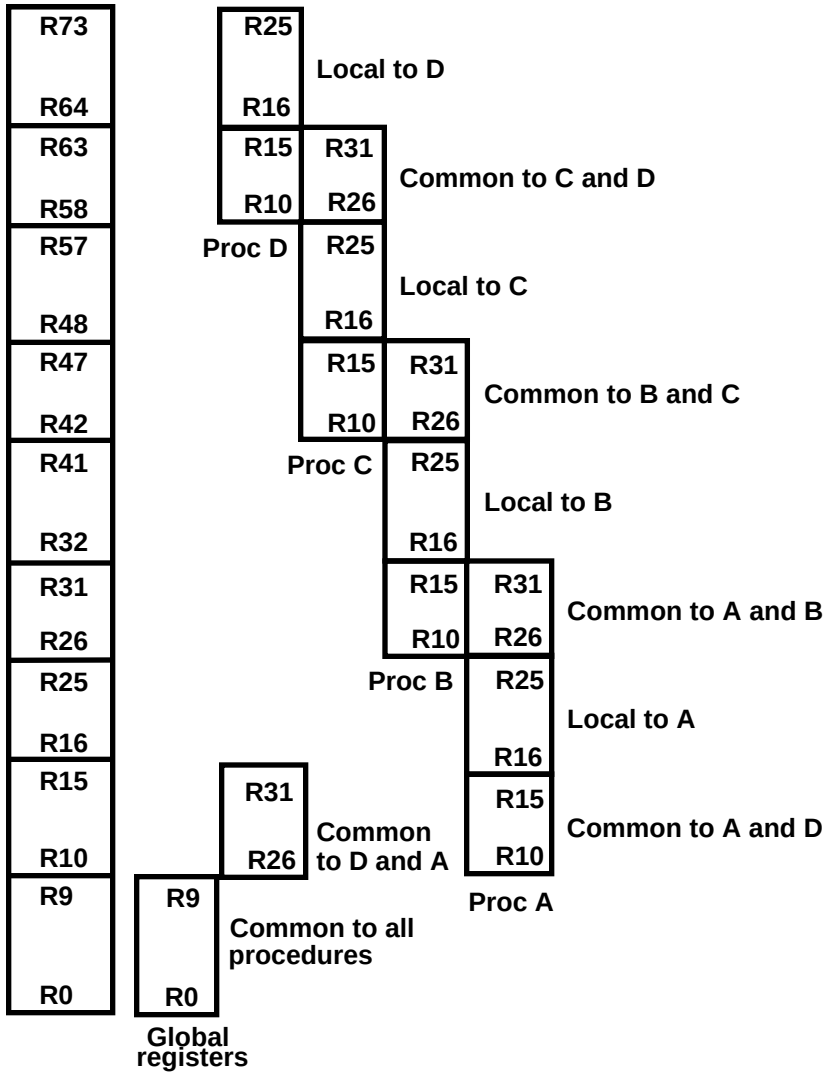
  - **A procedure call automatically switches the CPU to use a different window of registers, rather than saving registers in memory**

  - **Windows for adjacent procedures are overlapped to allow parameter passing**

CIRCULAR OVERLAPPED REGISTER WINDOWS



OVERLAPPED REGISTER WINDOWS



OVERLAPPED REGISTER WINDOWS

- There are three classes of registers:
  - Global Registers
    - » Available to all functions
  - Window local registers
    - » Variables local to the function
  - Window shared registers
    - » Permit data to be shared without actually needing to copy it
- Only one register window is active at a time
  - The active register window is indicated by a pointer
- When a function is called, a new register window is activated
  - This is done by incrementing the pointer
- When a function calls a new function, the high numbered registers of the calling function window are shared with the called function as the low numbered registers in its register window
- This way the caller's high and the called function's low registers overlap and can be used to pass parameters and results

OVERLAPPED REGISTER WINDOWS

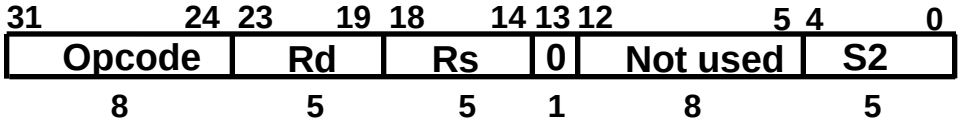
- In addition to the overlapped register windows, the processor has some number of registers, *G*, that are global registers
  - This is, all functions can access the global registers.
- The advantage of overlapped register windows is that the processor does not have to push registers on a stack to save values and to pass parameters when there is a function call
  - Conversely, pop the stack on a function return
- This saves
  - Accesses to memory to access the stack.
  - The cost of copying the register contents at all
- And, since function calls and returns are so common, this results in a significant savings relative to a stack-based approach

BERKELEY RISC I

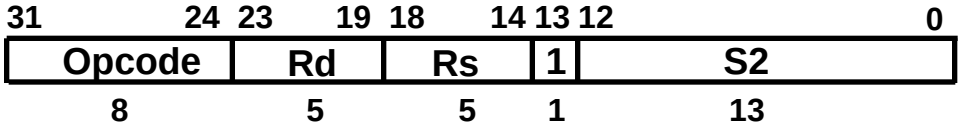
- 32-bit integrated circuit CPU
- 32-bit address, 8-, 16-, 32-bit data
- 32-bit instruction format
- total 31 instructions
- three addressing modes:
  register; immediate; PC relative addressing
- 138 registers
  10 global registers
  8 windows of 32 registers each

Berkeley RISC I Instruction Formats

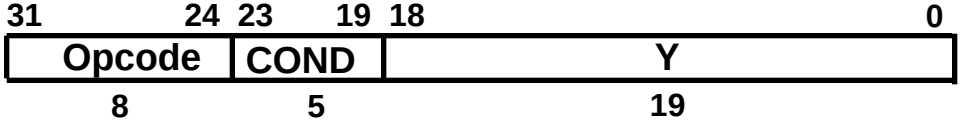
Regsiter mode: (S2 specifies a register)



Register-immediate mode (S2 specifies an operand)



PC relative mode



BERKELEY RISC I

- Register 0 was hard-wired to a value of 0.
- There are eight memory access instructions
  - Five load-from-memory instructions
  - Three store-to-memory instructions.

- The load instructions:

LDL	load long
LDSU	load short unsigned
LDSS	load short signed
LDBU	load byte unsigned
LDBS	load byte signed

  - Where long is 32 bits, short is 16 bits and a byte is 8 bits
- The store instructions:

STL	store long
STS	store short
STB	store byte

Berkeley RISC I

LDL	$R_d \leftarrow M[(R_s) + S_2]$	load long
LDSU	$R_d \leftarrow M[(R_s) + S_2]$	load short unsigned
LDSS	$R_d \leftarrow M[(R_s) + S_2]$	load short signed
LDBU	$R_d \leftarrow M[(R_s) + S_2]$	load byte unsigned
LDBS	$R_d \leftarrow M[(R_s) + S_2]$	load byte signed
STL	$M[(R_s) + S_2] \leftarrow R_d$	store long
STS	$M[(R_s) + S_2] \leftarrow R_d$	store short
STB	$M[(R_s) + S_2] \leftarrow R_d$	store byte

- Here the difference between the lengths is
  - A long is simply loaded, since it is the same size as the register (32 bits).
  - A short or a byte can be loaded into a register
    - » Unsigned - in which case the upper bits of the register are loaded with 0's.
    - » Signed - in which case the upper bits of the register are loaded with the sign bit of the short/byte loaded.

INSTRUCTION SET OF BERKELEY RISC I

Opcode	Operands	Register Transfer	Description
Data manipulation instructions			
ADD	Rs,S2,Rd	Rd ← Rs + S2	Integer add
ADDC	Rs,S2,Rd	Rd ← Rs + S2 + carry	Add with carry
SUB	Rs,S2,Rd	Rd ← Rs - S2	Integer subtract
SUBC	Rs,S2,Rd	Rd ← Rs - S2 - carry	Subtract with carry
SUBR	Rs,S2,Rd	Rd ← S2 - Rs	Subtract reverse
SUBCR	Rs,S2,Rd	Rd ← S2 - Rs - carry	Subtract with carry
AND	Rs,S2,Rd	Rd ← Rs ∧ S2	AND
OR	Rs,S2,Rd	Rd ← Rs ∨ S2	OR
XOR	Rs,S2,Rd	Rd ← Rs ⊕ S2	Exclusive-OR
SLL	Rs,S2,Rd	Rd ← Rs shifted by S2	Shift-left
SRL	Rs,S2,Rd	Rd ← Rs shifted by S2	Shift-right logical
SRA	Rs,S2,Rd	Rd ← Rs shifted by S2	Shift-right arithmetic
Data transfer instructions			
LDL	(Rs)S2,Rd	Rd ← M[Rs + S2]	Load long
LDSU	(Rs)S2,Rd	Rd ← M[Rs + S2]	Load short unsigned
LDSS	(Rs)S2,Rd	Rd ← M[Rs + S2]	Load short signed
LDBU	(Rs)S2,Rd	Rd ← M[Rs + S2]	Load byte unsigned
LDBS	(Rs)S2,Rd	Rd ← M[Rs + S2]	Load byte signed
LDHI	Rd,Y	Rd ← Y	Load immediate high
STL	Rd,(Rs)S2	M[Rs + S2] ← Rd	Store long
STS	Rd,(Rs)S2	M[Rs + S2] ← Rd	Store short
STB	Rd,(Rs)S2	M[Rs + S2] ← Rd	Store byte
GETPSW	Rd	Rd ← PSW	Load status word
PUTPSW	Rd	PSW ← Rd	Set status word

INSTRUCTION SET OF BERKELEY RISC I

Opcode	Operands	Register Transfer	Description
Program control instructions			
JMP	COND,S2(Rs)	PC ← Rs + S2	Conditional jump
JMPR	COND,Y	PC ← PC + Y	Jump relative
CALL	Rd,S2(Rs)	Rd ← PC, PC ← Rs + S2 CWP ← CWP - 1	Call subroutine and change window
CALLR	Rd,Y	Rd ← PC, PC ← PC + Y CWP ← CWP - 1	Call relative and change window
RET	Rd,S2	PC ← Rd + S2 CWP ← CWP + 1	Return and change window
CALLINT	Rd Call an interrupt pr.	Rd ← PC,CWP ← CWP - 1	
RETINT	Rd,S2	PC ← Rd + S2 CWP ← CWP + 1	Return from interrupt pr.
GTLPC	Rd	Rd ← PC	Get last PC

CHARACTERISTICS OF RISC

- RISC Characteristics
  - Relatively few instructions
  - Relatively few addressing modes
  - Memory access limited to load and store instructions
  - All operations done within the registers of the CPU
  - Fixed-length, easily decoded instruction format
  - Single-cycle instruction format
  - Hardwired rather than microprogrammed control

- Advantages of RISC
  - VLSI Realization
  - Computing Speed
  - Design Costs and Reliability
  - High Level Language Support

ADVANTAGES OF RISC

- **VLSI Realization**

  Control area is considerably reduced

Example:
RISC I: 6%
RISC II: 10%
MC68020: 68%
general CISCs: ~50%

⇒ RISC chips allow a large number of registers on the chip

- Enhancement of performance and HLL support
- Higher regularization factor and lower VLSI design cost

The GaAs VLSI chip realization is possible

- **Computing Speed**

- Simpler, smaller control unit ⇒ faster
- Simpler instruction set; addressing modes; instruction format
⇒ faster decoding
- Register operation ⇒ faster than memory operation
- Register window ⇒ enhances the overall speed of execution
- Identical instruction length, One cycle instruction execution
⇒ suitable for pipelining ⇒ faster

ADVANTAGES OF RISC

- **Design Costs and Reliability**

  - **Shorter time to design**
    ⇒ reduction in the overall design cost and reduces the problem that the end product will be obsolete by the time the design is completed

  - **Simpler, smaller control unit**
    ⇒ higher reliability

  - **Simple instruction format (of fixed length)**
    ⇒ ease of virtual memory management

- **High Level Language Support**

  - **A single choice of instruction**
    ⇒ shorter, simpler compiler

  - **A large number of CPU registers**
    ⇒ more efficient code

  - **Register window**
    ⇒ Direct support of HLL

  - **Reduced burden on compiler writer**