Appendix B: RISC (Reduced Instruction Set Computer) Processors:

RISC Features:

- Relatively small set of simple instructions
- Relatively few, simple addressing modes
- Fixed-length, easily decoded instruction format
- No instructions that operate directly on memory, all operations performed within internal registers of the CPU.
- Memory access only for load/store instructions (*load-store architecture*).
- One instruction per clock cycle (owing to pipelining)
- *Hardwired* rather than microprogrammed control unit

Other Characteristics:

Not all of the features listed below are included in all RISC processors, and CISC processors may also include some of these features:

- A large number of registers (128-256) (Register File)
- Use of overlapped register windows to speed up procedure call and return
- Instruction pipeline that can be optimized for instructions
- Harvard architecture
- Compiler support for efficient translation of high-level language programs into machine language programs

Overlapped Register Windows:

Without needing a stack (memory access), this structure is used in procedure calls

- to provide passing of parameters and
- to avoid the need for saving and restoring register values.

Even though the processor has many registers, the programmer can only use a limited number of these at any given moment.

This set of registers that can be active at any given time are called a **window**.

When the program calls (and returns from) a subroutine, the window changes. Thus, the programmer accesses different registers.

Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results. Local registers are used for local variables of the procedures.

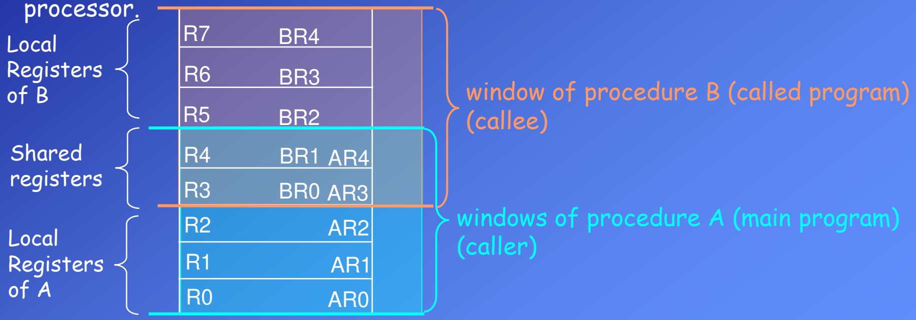
If there are n registers in a window, when writing programs, only registers R0 through Rn-1 are used.

However, as the window changes, these numbers correspond to different physical registers.

Not all RISC processors use this structure (e.g., the MIPS processor does not).

Example:

- In the example below, the processor has 8 registers. However, a window has 5 registers, so at any given time, only 5 of these can be active.
- In programs, only R0-R4 are used, but as the window changes, these end up corresponding to different registers.
- In A, when the programmer reaches R0, the programmer has reached R0 of the processor.
- In B, when the programmer reaches R0, the programmer has reached R3 of the processor.



There are also global registers with fixed numbers that all procedures access.

Determining the number of registers:

G	: Number of global registers
L	: Number of local registers in each window
C	: Number of registers common to two windows
W	: Number of windows

Window size = L + 2C + G (2*C because there are registers in common with the lower and upper window.)

Number of registers = (L+C)W + G

The window structure is arranged in a circular fashion.
If the processor has 4 windows, when the 4th procedure calls a 5th procedure, the 1st window (the oldest window, the one furthest back in the call nesting) is saved to memory.
Then, the 1st window is used by the 5th procedure.
When returning, the data in memory is restored to the relevant window.

Example:

In the next example, we give the register structure of a processor with a total of 74 registers and a windows size of 32 registers which supports procedure calls to a nesting depth of 4.

In this example, we assume that as procedures are called, subsequent windows are allocated registers with higher numbers. In real processors (RISC 1, SPARC), subsequent windows are allocated registers with lower numbers.

Example:

Total of 74 registers

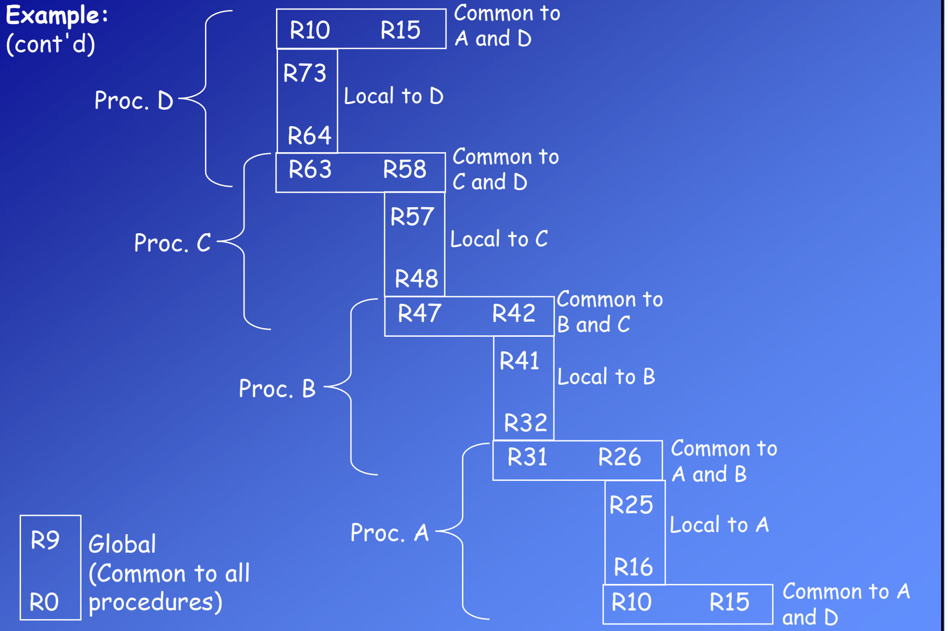
Global registers	: G = 10 (R0-R9) (common to all procedures)
R10-R73	: 64 registers divided into FOUR windows to accommodate procedures A-D. (W=4)
Local registers	: L = 10
Common registers	: 2*C = 6 + 6 =12
Window size	: L + 2C + G = 32 reg.

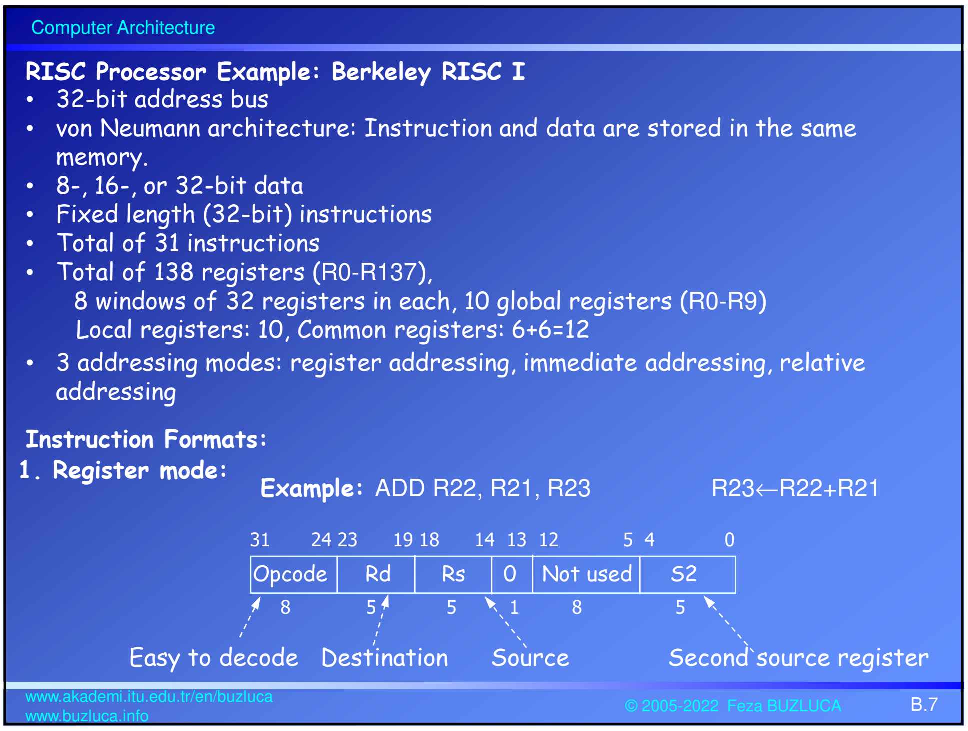
In this system, since there are 32 registers in a window, when writing a program in assembly language, each procedure uses register numbers R0 – R31.

Based on the location of the window, these numbers correspond to different physical registers.

See: exemplary program in B.13

Example:
(cont'd)





RISC Processor Example: Berkeley RISC I

- 32-bit address bus
- von Neumann architecture: Instruction and data are stored in the same memory.
- 8-, 16-, or 32-bit data
- Fixed length (32-bit) instructions
- Total of 31 instructions
- Total of 138 registers (R0-R137),
  8 windows of 32 registers in each, 10 global registers (R0-R9)
  Local registers: 10, Common registers: 6+6=12
- 3 addressing modes: register addressing, immediate addressing, relative addressing

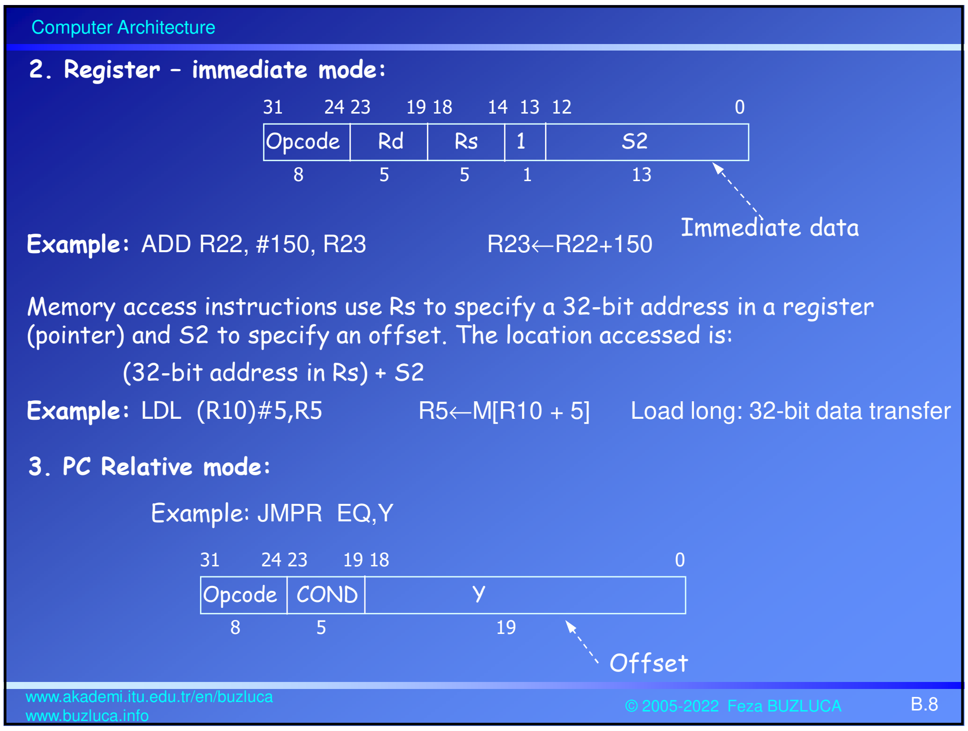
Instruction Formats:

1. Register mode:

Example: ADD R22, R21, R23 R23←R22+R21

31 … 24	23 … 19	18 … 14	13	12 … 5	4 … 0
Opcode	Rd	Rs	0	Not used	S2
8	5	5	1	8	5

Easy to decode Destination Source Second source register



2. Register - immediate mode:

31 … 24	23 … 19	18 … 14	13	12 … 0
Opcode	Rd	Rs	1	S2
8	5	5	1	13

Immediate data

Example: ADD R22, #150, R23 R23←R22+150

Memory access instructions use Rs to specify a 32-bit address in a register (pointer) and S2 to specify an offset. The location accessed is:

(32-bit address in Rs) + S2

Example: LDL (R10)#5,R5 R5←M[R10 + 5] Load long: 32-bit data transfer

3. PC Relative mode:

Example: JMPR EQ,Y

31 … 24	23 … 19	18 … 0
Opcode	COND	Y
8	5	19

Offset

Usage of Berkeley RISC I Instructions

Register R0 contains all 0's, so it can be used in any field to specify a zero quantity. By using register R0, which always contains 0's (zeros), it is possible to transfer the contents of one register or a constant into another register.

ADD	R0, R21, R22	R22←R21	(Move)
ADD	R0, #150, R22	R22←150	(Immediate load)
ADD	R22, #1, R22	R22←R22 +1	(Increment)

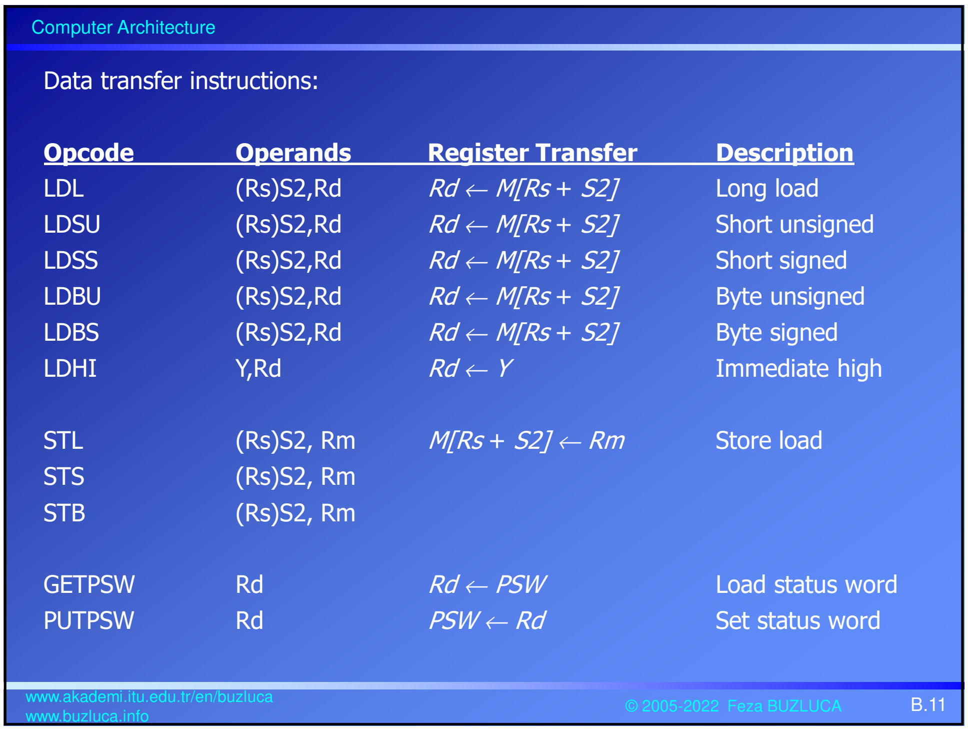
The load and store instructions move data between a register and memory.

LDL	(R22)#150,R5	R5←M[R22 +150]	Load long: 32-bit data transfer
LDL	(R22)#0,R5	R5←M[R22]	
LDL	(R0)#500,R5	R5←M[500]	

Instruction Set of Berkeley RISC I

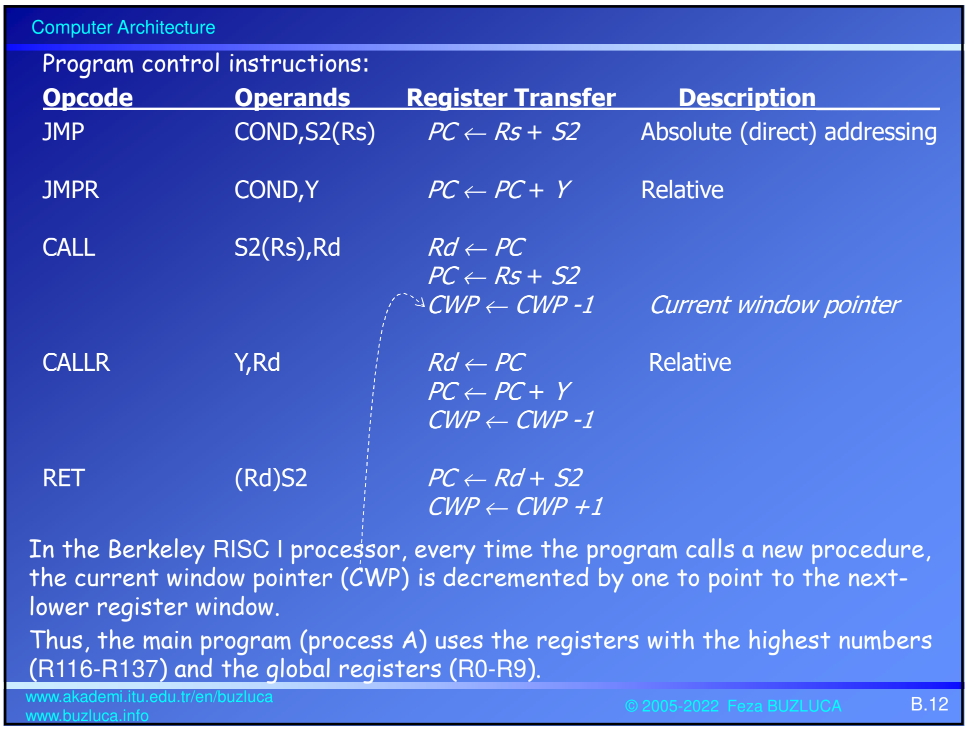
Data manipulation instructions:

Opcode	Operands	Register Transfer
ADD	Rs,S2,Rd	*Rd* ← *Rs* + S2
ADDC	Rs,S2,Rd	*Rd* ← *Rs* + S2 + carry
SUB	Rs,S2,Rd	*Rd* ← *Rs* - S2
SUBC	Rs,S2,Rd	*Rd* ← *Rs* - S2 - carry
SUBR	Rs,S2,Rd	*Rd* ← *S2* - *Rs*
SUBCR	Rs,S2,Rd	*Rd* ← *S2* - *Rs* - carry
AND	Rs,S2,Rd	*Rd* ← *Rs* ∧ S2
OR	Rs,S2,Rd	*Rd* ← *Rs* ∨ S2
XOR	Rs,S2,Rd	*Rd* ← *Rs* ⊕S2
SLL	Rs,S2,Rd	*Rd* ← *Rs* shifted by S2
SRL	Rs,S2,Rd	*Rd* ← *Rs* shifted by S2
SRA	Rs,S2,Rd	*Rd* ← *Rs* shifted by S2



Data transfer instructions:

Opcode	Operands	Register Transfer	Description
LDL	(Rs)S2,Rd	*Rd ← M[Rs + S2]*	Long load
LDSU	(Rs)S2,Rd	*Rd ← M[Rs + S2]*	Short unsigned
LDSS	(Rs)S2,Rd	*Rd ← M[Rs + S2]*	Short signed
LDBU	(Rs)S2,Rd	*Rd ← M[Rs + S2]*	Byte unsigned
LDBS	(Rs)S2,Rd	*Rd ← M[Rs + S2]*	Byte signed
LDHI	Y,Rd	*Rd ← Y*	Immediate high
STL	(Rs)S2, Rm	*M[Rs + S2] ← Rm*	Store load
STS	(Rs)S2, Rm		
STB	(Rs)S2, Rm		
GETPSW	Rd	*Rd ← PSW*	Load status word
PUTPSW	Rd	*PSW ← Rd*	Set status word

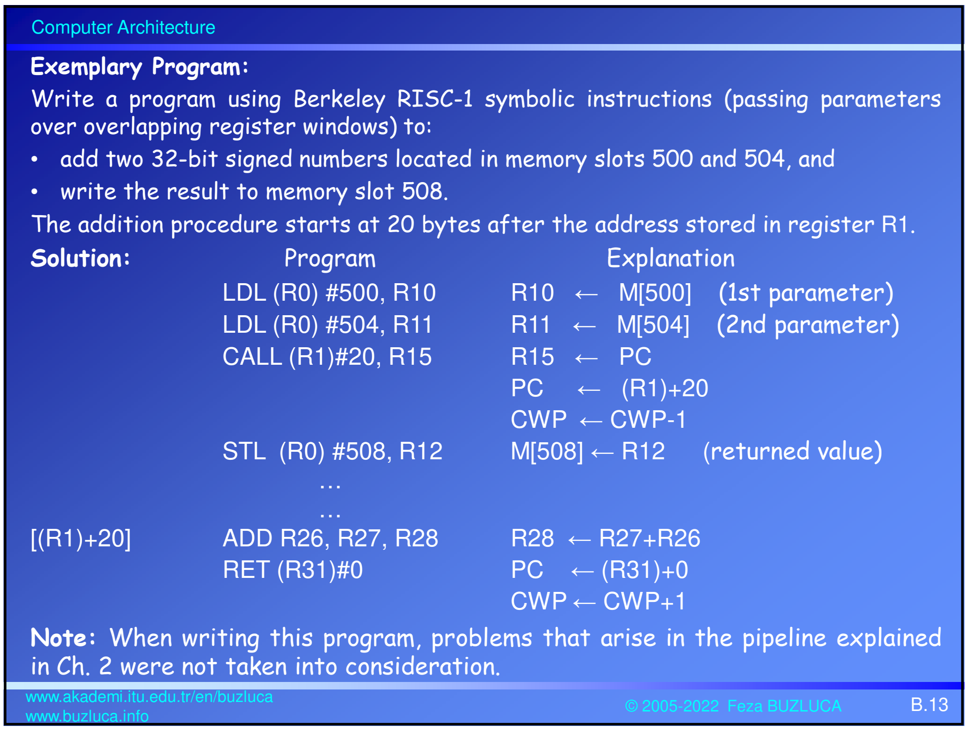


Program control instructions:

Opcode	Operands	Register Transfer	Description
JMP	COND,S2(Rs)	*PC ← Rs + S2*	Absolute (direct) addressing
JMPR	COND,Y	*PC ← PC + Y*	Relative
CALL	S2(Rs),Rd	*Rd ← PC* *PC ← Rs + S2* *CWP ← CWP -1*	*Current window pointer*
CALLR	Y,Rd	*Rd ← PC* *PC ← PC + Y* *CWP ← CWP -1*	Relative
RET	(Rd)S2	*PC ← Rd + S2* *CWP ← CWP +1*	

In the Berkeley RISC I processor, every time the program calls a new procedure, the current window pointer (CWP) is decremented by one to point to the next-lower register window.

Thus, the main program (process A) uses the registers with the highest numbers (R116-R137) and the global registers (R0-R9).



Exemplary Program:

Write a program using Berkeley RISC-1 symbolic instructions (passing parameters over overlapping register windows) to:

- add two 32-bit signed numbers located in memory slots 500 and 504, and
- write the result to memory slot 508.

The addition procedure starts at 20 bytes after the address stored in register R1.

Solution:

```
                Program                     Explanation
                LDL (R0) #500, R10          R10  ←  M[500]   (1st parameter)
                LDL (R0) #504, R11          R11  ←  M[504]   (2nd parameter)
                CALL (R1)#20, R15           R15  ←  PC
                                            PC   ←  (R1)+20
                                            CWP ← CWP-1
                STL  (R0) #508, R12         M[508] ← R12     (returned value)
                         …
                         …
[(R1)+20]       ADD R26, R27, R28           R28  ← R27+R26
                RET (R31)#0                 PC   ← (R31)+0
                                            CWP ← CWP+1
```

Note: When writing this program, problems that arise in the pipeline explained in Ch. 2 were not taken into consideration.