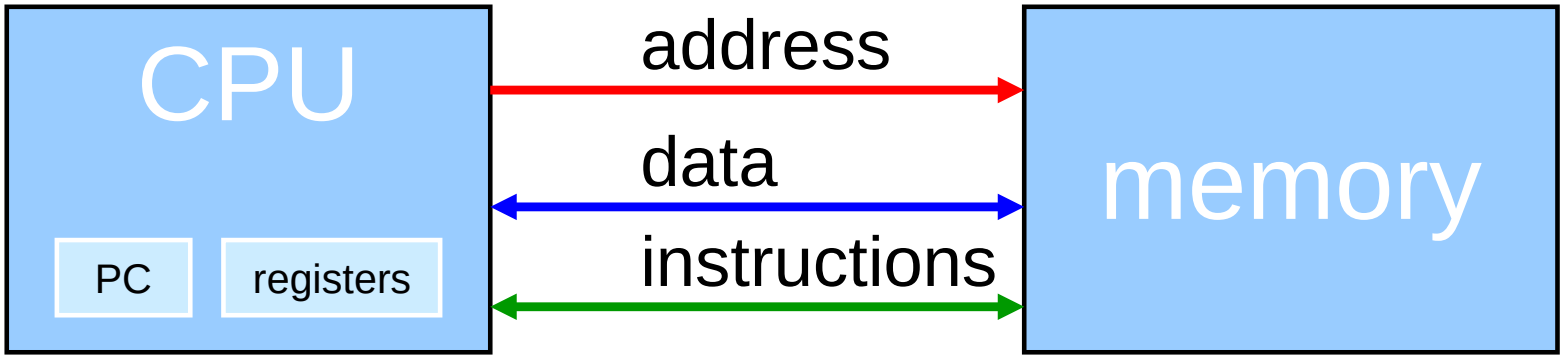
Architecture of Digital Systems II

2 Microprocessor Instruction Sets

Computer Architecture Basics

In general, a computing system contains a central processing unit (CPU) and a memory:



The memory holds both the program (instructions) and the data. Every memory cell has a unique address.

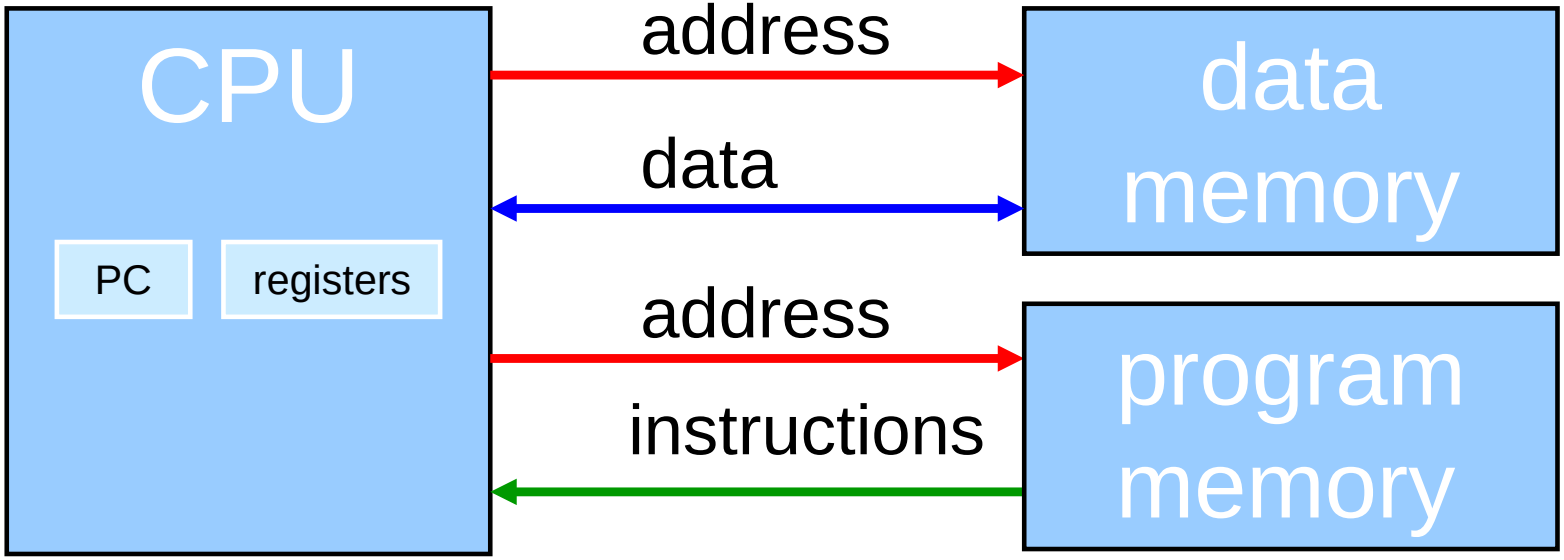
The CPU has several internal registers for storing data internally. One important register is the program counter (PC) which holds the memory address of the next instruction.

Architecture categories

There are two basic types of computer architectures:

A computer whose memory holds both data and instructions has a von-Neumann architecture. This is the most widely used architecture today.

A computer organization with separate memories for data and program is called a Harvard architecture:



Harvard Architecture

Separation of program and data memories provides higher performance, especially in digital signal processing.
DSPs (digital signal processors) are mostly built in Harvard architecture.
The two separate memory ports – two data busses and two address busses – provide a higher memory bandwidth.
(Both memories can be accessed simultaneously.)

Note:
Today, even general-purpose processors with an external von-Neumann architecture may be implemented as Harvard machines at the cache level, internally. They have separate data and instruction caches. This is, however, invisible to the programmer.

Instruction Set and Programming Model

The instruction set of a processor defines the interface between the software and the hardware. It reflects the capabilities of the hardware.

Characteristics of instruction sets are:

- types of operations supported
- numbers of operands
- types of operands: register or memory
- addressing modes
- length of instructions

The instruction set greatly depends on the programming model of the processor: this is the set of registers available for use by programs.

CISC vs. RISC

Early computer architectures were complex-instruction-set computers (CISCs):

- instruction set comprises a great variety of instructions
- some instructions are very powerful and complex
- instructions are of varying length
- operands may be in memory or in registers
- programs are more compact

(CISC vs. RISC)

Today, high-performance processors are designed as reduced-instruction-set computers (RISCs):

- instruction set contains only "simple" instructions
- load/store architecture

RISC advantages:

- uniform instruction format
  → one instruction execution per cycle is possible
- load/store architecture separates slow memory access from fast computations (instruction sequences can be optimized by compiler)
- reduced instruction complexity
  → faster hardware, higher clock frequency
  → processor can be easily pipelined!
  → one (or more) instructions can be executed per cycle

Processors for Embedded Computing

In the following, we will consider the instruction sets of two specific microprocessors. These processors are examples of CPUs typically employed in embedded systems.

ARM is a typical general-purpose RISC processor family (like MIPS, SPARC and others).

SHARC is DSP architecture tuned to the specific needs of digital signal processing applications.

Both are processor *families*, meaning that there are different implementations of their instruction sets available, offering a spectrum of performance and configurations to serve different application needs.

ARM Architecture Characteristics

There are different versions of the ARM architecture. We consider version ARM7 here.
The ARM architecture has the following basic characteristics:

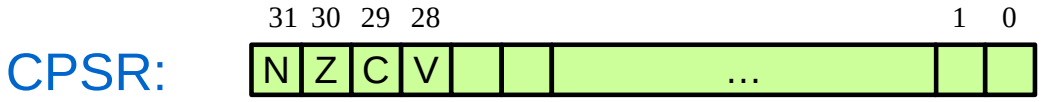
- load/store architecture
- standard word size: 32 bits
- address length: 32 bits (byte addressable)
- operations for loading/storing half words and bytes
- configurable for little-endian and big-endian addressing modes

ARM Programming Model

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 (PC)

There are 16 general-purpose registers, named r0 to r15.
Register r15 serves as the program counter (PC).

The CPSR is the current program status register. Its four most significant bits contain status information. They may be set during an ALU instruction.



CPSR status bits (condition flags):

- **N**: negative
- **Z**: zero
- **C**: carry
- **V**: overflow

ARM Instruction Set (1)

Arithmetic instructions

Operation	Assembler Examples	Register Transfer
Addition	ADD r0,r1,r2	r0=r1+r2 (register operand)
	ADD r0,r1,#5	r0=r1+5 (immediate operand)
	ADDS r0,r1,#5	r0=r1+5; updates N,C,Z,V
Add with carry	ADC r0,r1,r2	r0=r1+r2+C
Subtract	SUB r0,r1,r2	r0=r1-r2
Subtract with carry	SBC r0,r1,r2	r0=r1-r2-NOT(C)

- Instructions may provide immediate operands (see second example).
- Instructions may be configured to update the CPSR (by appending the letter "S“ to the instruction mnemonic).

ARM Instruction Set (2)

(Arithmetic instructions, more examples)

Operation	Example	Register Transfer
parallel half-word addition	ADD16 r0,r1,r2	r0[31:16]=r1[31:16]+r2[31:16] r0[15:0]=r1[15:0]+r2[15:0]
multiply	MUL r0,r1,r2	r0=r1*r2[31:0]
multiply/accumulate	MLA r0,r1,r2,r3	r0=(r1*r2)[31:0]+r3

ARM Instruction Set (3)

Logical instructions

Operation	Example	Register Transfer
bitwise AND	AND r0,r1,r2	r0=r1 AND r2
bitwise OR	ORR r0,r1,r2	r0=r1 OR r2
bitwise XOR	EOR r0,r1,r2	r0=r1 XOR r2
bit clear	BIC r0,r1,r2	r0=r1 AND (NOT r2)

Move instructions

Operation	Example	Register Transfer
Move	MOV r0,r1	r0=r1
Move negated	MVN r0,r1	r0=NOT(r1)

ARM Instruction Set (4)

Load/Store instructions

Operation	Example	Register Transfer
Load word	LDR r0,[r1]	r0=[r1] *(register-indirect addressing)*
Load half-word	LDRH r0,[r1]	r0=zeroextend([r1][15:0])
Load half-word signed	LDRSH r0,[r1]	r0=signextend([r1][15:0])
Load byte	LDRB r0,[r1]	r0=zeroextend([r1][7:0])
Load byte signed	LDRSB r0,[r1]	r0=signextend([r1][7:0])
Store word	STR r0,[r1]	[r1]=r0
Store half-word	STRH r0,[r1]	[r1][15:0]=r0[15:0]
Store byte	STRB r0,[r1]	[r1][7:0]=r0[7:0]

ARM Instruction Set (4)

Addressing modes

Addressing mode	Examples	Register Transfer
Register-Indirect	LDR r0,[r1]	r0=[r1]
Base-plus-offset	LDR r0,[r1,r2]	r0=[r1+r2]
	LDR r0,[r1,#4]	r0=[r1+4] *(immediate offset)*
Auto-indexing	LDR r0,[r1,r2]!	first r1=r1+r2, then r0=[r1]
	LDR r0,[r1,#16]!	first r1=r1+16, then r0=[r1]
Post-indexing	LDR r0,[r1],r2	first r0=[r1], then r1=r1+r2
	LDR r0,[r1],#4	first r0=[r1], then r1=r1+4

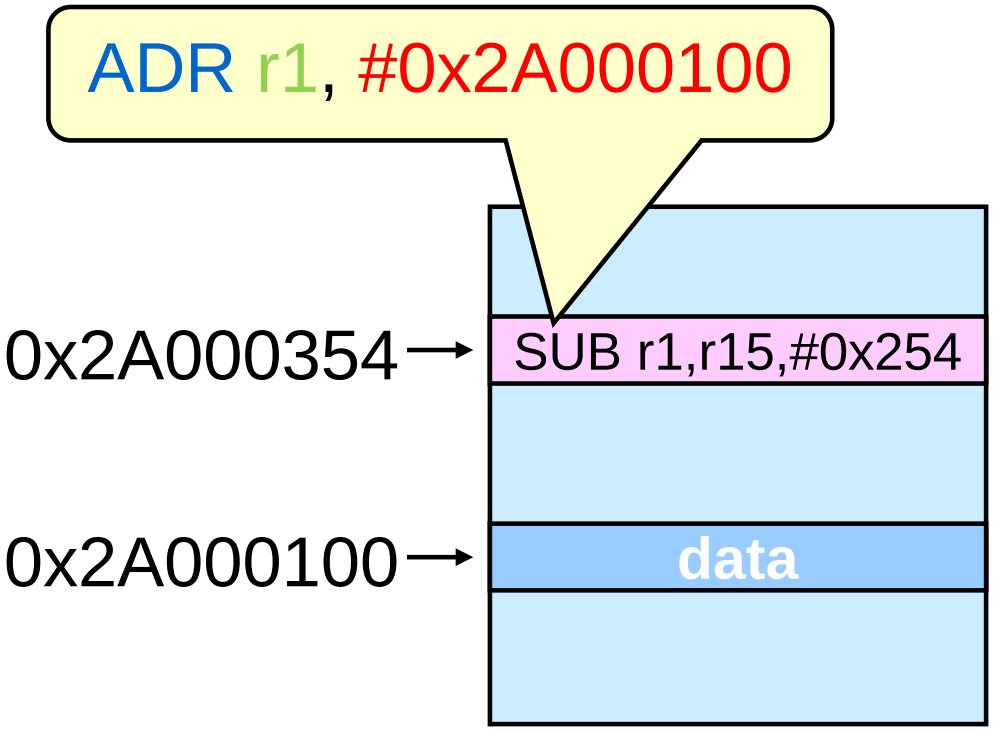
Auto-indexing and post-indexing addressing modes can be used to iterate through data arrays.

ARM Instruction Set (5)

PC-relative addressing

By using r15 (the PC) as base register and immediates as offset, memory addresses can be computed relative to the current program location.

The ARM programming system provides the pseudo-operation ADR:



ADR r*X*, *addr*

(set rX to hold address *addr*)

The assembler replaces the pseudo-operation with an appropriate instruction.

Data Representation in Assembler Programs

Data is stored at certain memory locations and in blocks of certain sizes. Instead of directly handling memory addresses explicitly, we associate symbols (variable names) for these memory locations. In assembler notation, symbols are defined using labels. Memory space is reserved using declaration directives.

Examples:

```
c:      DCB 65          ; allocate a byte and initialize it with 'A'
sum:    DCW 7           ; allocate a half-word and initialize it with 7
year:   DCD 2008        ; allocate a word (32 bits) and initialize with 2008
msg:    DCS "ERROR"     ; allocate a string (sequence of bytes)
```

The labels can be used as addresses in assembler instructions.

Example:

```
        ADR r1,year     ; load address of year variable into r1
```

ARM Instruction Set (5)

Shifting

Shift and rotate commands are not separate instructions but can be applied to the set of arithmetic and logical instructions. If applied, always the second source operand is modified.

Modifier	Operation performed on the second operand
LSL	Logical shift left (filling with zeros)
LSR	Logical shift right (filling with zeros)
ASL	Arithmetic shift left (filling with zeros)
ASR	Arithmetic shift right (filling with sign)
ROR	Rotate right
RRX	Rotate right through carry bit C

Examples:

```
ADD r0,r1,r2,ASL #2      ; r0=r1+(r2<<2)  -- shift by immediate
SUB r0,r1,r2,LSL r3      ; r0=r1-(r2<<r3)  -- shift by register
```

Control Flow

In software programs, we need means for changing the *control flow*, i.e., the order, in which program statements are executed.

Examples:

Conditional execution:

```
if (a < b) {
        ...
} else {
        ...
}
```

Loops:

```
while (a < b) {
        ...
        a = a+1;
}
```

Subroutines:

```
        int do_something(int a)
                return 42;
        }
...
        int main()
        {
                int x = do_something(5);
        }
```

ARM Instruction Set (6)

In ARM, the basic mechanism for changing the control flow is the branch instruction. Branches are PC-relative.

Operation	Examples	Register Transfer
Branch	B #100	r15=r15+(#100)*4
	(the offset is in 4-byte words)	

The branch can be made *conditional*, to allow to change the execution path through the program if a certain condition is fulfilled.

Example:

```
up:     BEQ down      ; if Z flag is set (ALU result was 0), branch!
        ....
        B up          ; unconditional branch
down:   ...           ; branch target
```

ARM Instruction Set (7)

Condition codes

Condition	Code	Status Flags Checked
Equals zero	EQ	Z=1
Not equal to zero	NE	Z=0
Carry set	CS	C=1
Carry clear	CC	C=0
Minus	MI	N=1
Plus	PL	N=0
Overflow	VS	V=1
No overflow	VC	V=0
Unsigned higher	HI	C=1 and Z=0
Unsigned less than or equal	LS	C=0 or Z=1
Signed greater than or equal	GE	N=V
Signed Less than	LT	N≠V
Signed greater than	GT	Z=0 and N=V
Signed less than or equal	LE	Z=1 or N≠V

ARM Instruction Set (8)

Comparison instructions

The comparison instructions modify only the NZCV bits in the CPSR status register and leave the other registers unaffected.

Operation	Example	Register Transfer
Compare	CMP r0,r1	sets NZCV bits for r0–r1
Negated Compare	CMN r0,r1	sets NZCV bits for r0+r1
Bit-wise test	TST r0,r1	sets NZCV bits for r0 AND r1
Bit-wise negated test	TEQ r0,r1	sets NZCV bits for r0 EOR r1

Comparison instructions are used frequently, to implement conditional execution.

ARM Instruction Set (9)

Example: if-then-else construct in ARM assembler

C code:

```
if (r0 < r1) {
        r2 = r3+r4;  /* then-branch */
} else {
        r2 = r3-r4;  /* else-branch */
}
/* end-if */
```

ARM code:

```
            CMP r0,r1
            BGE elsebr
thenbr      ADD r2,r3,r4
            B endif
elsebr      SUB r2,r3,r4
endif       ...
```

ARM Instruction Set (10)

Conditional instruction execution

ARM allows not only branches to be executed conditionally. The condition code can be added to *any* instruction.

Example: revised if-then-else construct in ARM assembler

C code:

```
if (r0 < r1) {
        r2 = r3+r4;  /* then-branch */
} else {
        r2 = r3-r4;  /* else-branch */
}
/* end-if */
```

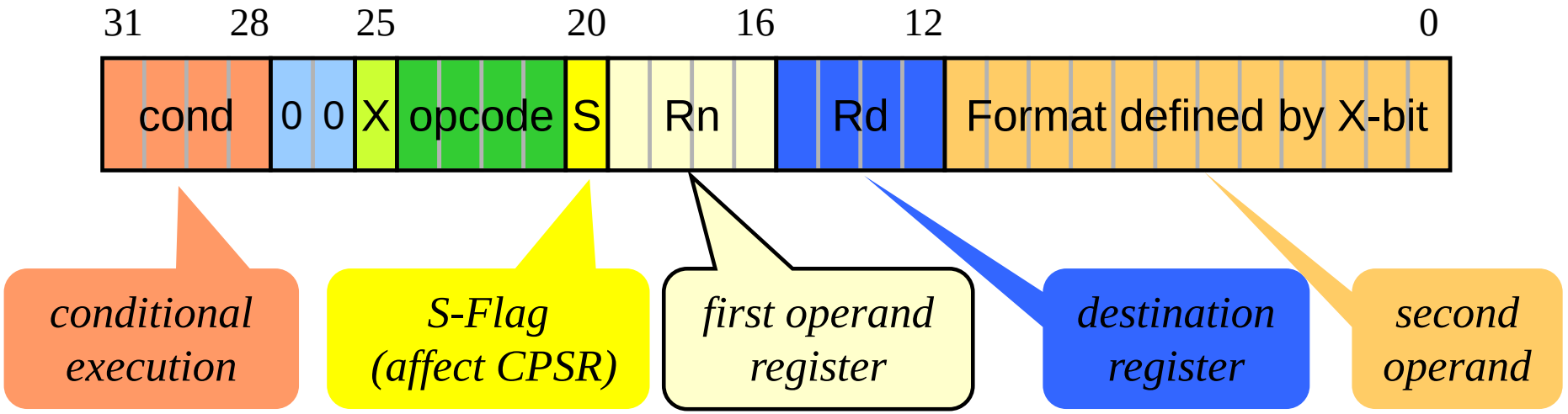
ARM code:

```
        CMP r0,r1
thenbr  ADDLT r2,r3,r4
elsebr  SUBGE r2,r3,r4
endif   ...
```

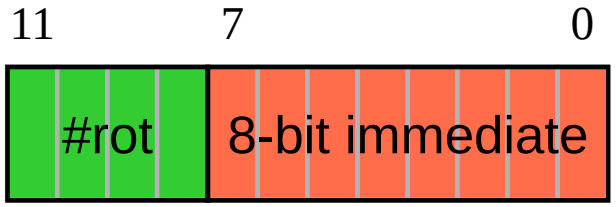
The ADD and SUB instructions are executed only if the condition predicates LT or GE are true.

ARM Instruction Format – Example

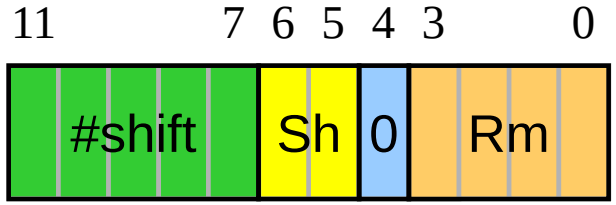
The ARM instruction format allows many instruction variants.
Example: Format for data processing instructions such as ADD:



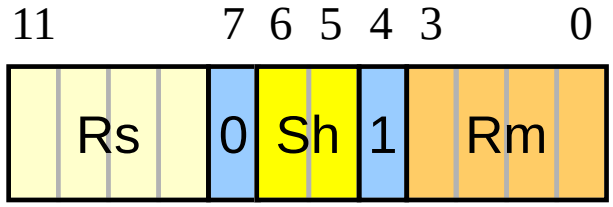
X = 1:



X = 0:



X = 0:



ARM Instruction Set (11)

Subroutine support

The ARM architecture provides the branch-and-link instruction to implement procedure calls. It stores the return address (PC value of instruction following the branch) in register r14, then jumps to the subroutine address.

Operation	Example	Register Transfer
Branch-and-link	BL target	r14=address of next instruction (return address); r15=target

Example (flawed!):

C code:

```
void f1() {
        ...
        f2();
        ...
}
```

ARM code:

```
f1      ...             ; begin f1
        BL f2           ; r14=f2ra
f2ra    ...
f2      ...             ; begin f2
        MOV r15,r14     ; return from f2
```

ARM Instruction Set (12)

In order to implement nested function calls we need to build a function call stack. The stack contains one activation record for each active procedure, containing procedure parameters and the return address. Register r13 is used as the stack pointer.

Example (revised):

C code:

```
void f1() {
        ...
        f2();
        ...
}
```

ARM code:

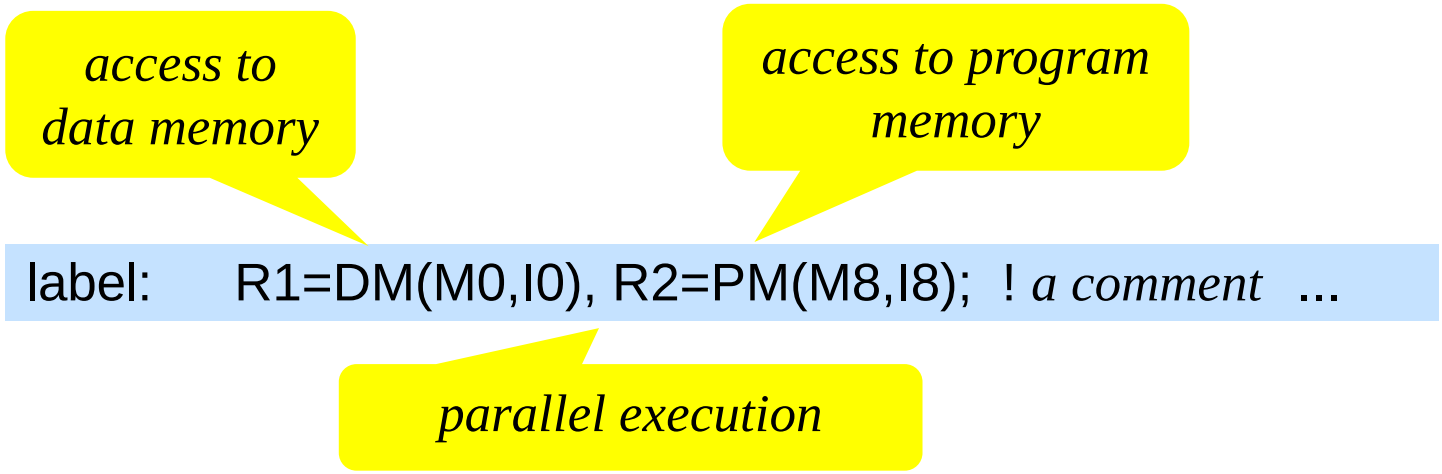
```
f1      ...                     ; begin f1
        STR r14,[r13,#-4]!      ; save f1's return address
                                ; on the stack
        BL f2                   ; r14=f2ra
f2ra    LDR r14,[r13],#4        ; restore f1's return address
        ...
f2      ...                     ; begin f2
        MOV r15,r14             ; return from f2
```

SHARC: Example of a Digital Signal Processor

In the following we will study an example of a DSP family: SHARC. The architecture is tuned for application in digital signal processing:

- Harvard architecture, load-store
- program and data memory on-chip
- floating-point hardware
- powerful arithmetic instructions

These features are reflected in the instruction set and the programming model. Example of a SHARC instruction:



SHARC: Architectural Characteristics

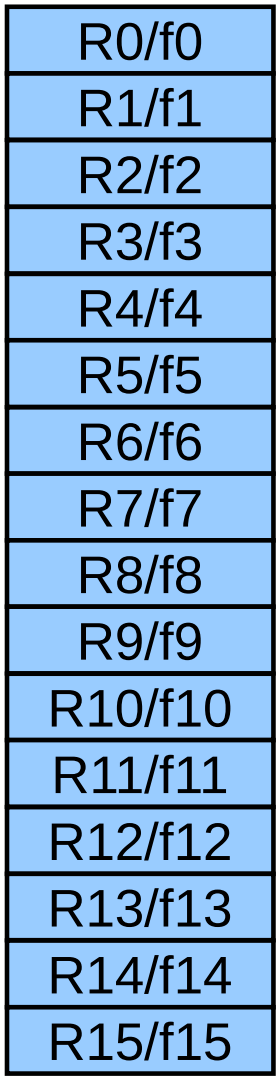
SHARC = Super Harvard Architecture Computer

is a modified Harvard architecture. The program memory and the data memory are separated as usual. However, data may also be placed in program memory. This allows for two memory accesses in a single instruction. If the instruction is in the instruction cache, the two simultaneous accesses can be executed in a single cycle.

Memory organization

- Program memory (PM) and data memory (DM) are on-chip
- Instruction word size: 48 bits
- Basic data types:
  - 32-bit integers
  - 32-bit floating point (IEEE 754 single-precision)
  - 40-bit floating point (single-extended precision, 31-bit mantissa)

SHARC Programming Model (1)



The SHARC programming model is quite complex. We will only give an overview here.

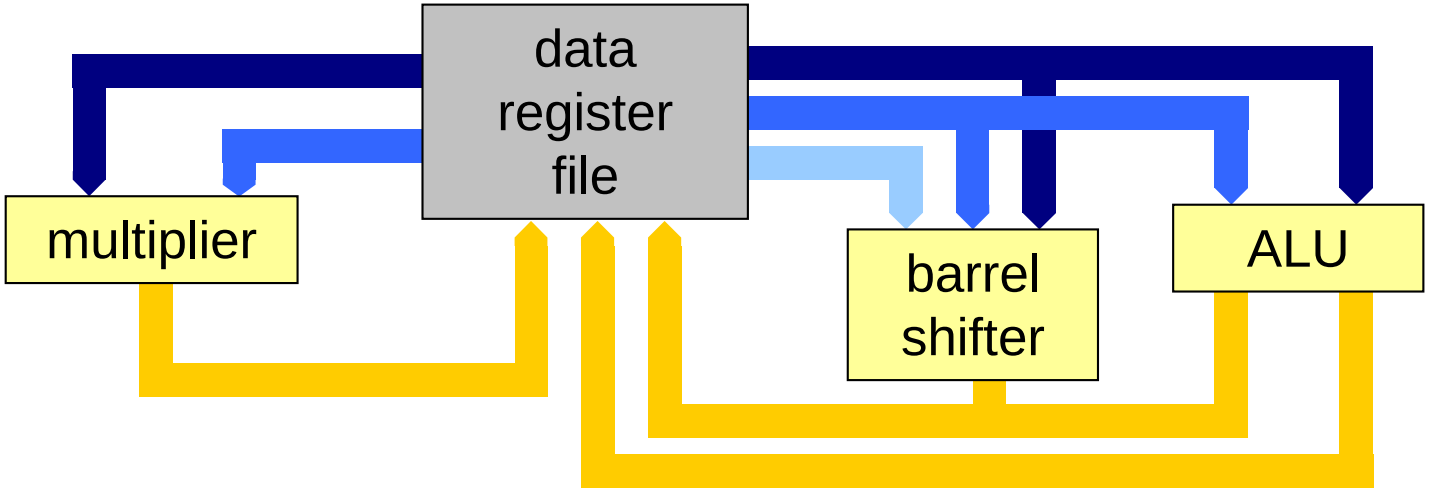
The 16 primary data registers are named R0..R15 in integer operations and f0..f15 in floating-point operations. The registers are 40 bits wide. In integer operations, only 32 bits are used.

There are three important status/mode registers:

ASTAT	arithmetic status register
STKY	sticky status flag status register
MODE1	mode control register

SHARC Data Path (1)

A SHARC CPU has one or more processing elements consisting of a multiplier, a barrel shifter and an ALU.



In the status registers ASTAT and STKY, there are fields for each of the functional units.

SHARC Data Path (2)

The functional units set status flags in **ASTAT** register:

ALU
AZ: ALU zero
AV: ALU overflow
AN: ALU negative
AC: ALU carry
...

Multiplier
MN: multiplier negative
MV: multiplier overflow
MU: multiplier underflow
...

Shifter
SZ: zero
SV: overflow
SS: input sign

ALU and multiplier also set status flags in the **STKY** register. These flags remain set until explicitly cleared. This allows to evaluate conditions after a sequence of arithmetic operations.

ALU
AOS: ALU fixed point overflow
AUS: ALU floating point underflow
AVS: ALU floating point overflow
...

Multiplier
MOS: multiplier fixed point overflow
MUS: multiplier floating point underflow
MVS: multiplier floating point overflow

SHARC Data Path (3)

Bits in the **MODE1** register control general operational modes of the functional units.

Control bit	Function	
TRUNC	1=truncate,	0=round to nearest
RND32	1=round to 32 bits,	0=round to 40 bits
ALUSAT	perform saturation arithmetic	

A note on saturation arithmetic:

In some DSP applications, it is preferable to *saturate* any overflowing or underflowing fixed-point computation result instead of the usual *wrapping* around the numeric range. When saturating, an overflow will result in the maximum value of the range, an underflow will result in the minimum value of the range.

SHARC Instruction Set (1) – Fixed-point ALU ops.

Operation	Assembler	Remarks
Add	Rn=Rx+Ry	
Add with carry	Rn=Rx+Ry+AC	
Add carry	Rn=Rx+AC	
Add borrow	Rn=Rx+!AC	
Increment	Rn=Rx+1	*1 is not an immediate operand*
Subtract	Rn=Rx-Ry	
Subtract with borrow	Rn=Rx-Ry+!AC	
Decrement	Rn=Rx-1	
Compare	COMP(Rx,Ry)	*Subtract without writing result*
Average	Rn=(Rx+Ry)/2	
Negate	Rn=–Rx	
Absolute value	Rn=ABS Rx	
Copy Rx to Rn	Rn=PASS Rx	
Logical AND	Rn=Rx AND Ry	
Logical OR	Rn=Rx OR Ry	
Logical XOR	Rn=Rx XOR Ry	
Logical negate	Rn=NOT Rx	
Minimum	Rn=MIN(Rx,Ry)	
Maximum	Rn=MAX(Rx,Ry)	
Clip	Rn=CLIP Rx by Ry	*Clip Rx within range* [*-Ry,Ry*]

SHARC Instruction Set (2) – Floating-Point ALU ops.

Operation	Assembler	Remarks
Add	Fn=Fx+Fy	
Subtract	Fn=Fx-Fy	
Absolute value of sum	Fn=ABS(Fx+Fy)	
Absolute value of difference	Fn=ABS(Fx-Fy)	
Average	Fn=(Fx+Fy)/2	
Compare	COMP(Fx,Fy)	
Negate	Fx=-Fx	
Copy Fx to Fn	Fn=Pass Fx	
Round	Fn=RND Fx	*rounds* Fx *to 32 bit boundary*
Scale exponent	Fn=SCALB FX by Ry	*add Ry to exponent of Fx*
Extract Mantissa	Rn=MANT Fx	*result is unsigned integer*
Extract Exponent	Rn=LOGB Fx	*result is signed (two's complement)*
Convert floating-point to integer	Rn=FIX Fx, Rn=TRUNC Fx	
Convert integer to floating-point	Fn=FLOAT Rx, Fn=FLOAT Rx by Ry	
Create seed for reciprocal	Fn=RECIPS Fx	*begin reciprocal computation*
Create seed for recipr. SQRT	Fn=RSQRTS Fx	*begin computation of recipr. square root*
Copy sign of Fy to sign of Fx	Fx COPYSIGN Fy	
Minimum	Fn=MIN(Fx,Fy)	
Maximum	Fn=MAX(Fx,Fy)	
Clip	Fn=CLIP Fx by Fy	*Clip Rx within range [-Ry,Ry]*
…		

SHARC Instruction Set (3) - Multiplier

The multiplier is used for both fixed-point and floating-point computations. The multiplication result can be stored in two 80-bit registers, MRF and MRB. Fixed-point numbers may be interpreted as integers or as fractional numbers (decimal point to the right of the left-most digit).

Operation	Assembler	Remarks
Multiply	Rn=Rx*Ry (mod2)	*mod2: see below*
	MRz=Rx*Ry (mod2)	z=F|B
	Fn=Fx*Fy	
Multiply/accumulate	Rn=MRx+Rx*Ry (mod2)	
	MRz=MRx+Rx*Ry (mod2)	
	Rn=MRx–Rx*Ry (mod2)	
	MRz=MRx–Rx*Ry (mod2)	
Transfer	MRmz=Rn	m=0|1|2, z=F|B
	Rn=MRmz	
Initialize	MRz=0	

The modifier mod2 *specifies the type of the two source operands:*

S=signed, U=unsigned, I=integer, F=fractional

Example: mod2=SUI: *first operand* signed, *second operand* unsigned, integers.

SHARC – Parallelism within Instructions

SHARC can execute several operations in parallel, i.e., in a single instruction:

- fixed-point multiply/accumulate + an add, subtract or average
- floating-point multiplication + floating-point ALU operation
- multiplication + dual add/subtract

(A dual add/subtract is the computation of the sum and the difference of the same two operands).

Example of a multi-operation instruction:

R6 = R0 * R4, R9 = R8 + R12, R10 = R8 – R12

Restrictions apply on the sources of the operands in parallel instructions.

SHARC Instruction Set (4) – Shifter Operations

The shifter supports a large number of operations. Some examples:

Operation	Assembler	Remarks
Logical shift	Rn=LSHIFT Rx by Ry	*Ry is signed, positive value means "left"*
	Rn=LSHIFT Rx by #5	*shift by an immediate*
		(many instructions may shift by immediate)
Logical shift+OR	Rn=Rn OR LSHIFT Rx by Ry	*first shift, then logical OR*
Arithmetic shift	Rn=ASHIFT Rx by Ry	
Rotate	ROT Rx by Ry	
Clear bit	Rn=BCLR Rx by Ry	*Ry is bit position in Rx*
Set bit	Rn=BSET Rx by Ry	*Ry is bit position in Rx*
Toggle bit	Rn=BTGL Rx by Ry	*Ry is bit position in Rx*
Leading zeros	Rn=LEFTZ Rx	*get number of leading zeros*
Leading ones	Rn=LEFTO Rx	*get number of leading ones*

Logical shifts fill with zeros, arithmetic shifts copy sign bits.

SHARC Addressing (1)

SHARC has a modified Harvard architecture: data may also be stored in program memory.

Operation	Assembler
Read from program memory	Rn=PM(<address>)
Read from data memory	Rn=DM(<address>)
Write to program memory	PM(<address>)=Rn
Write to data memory	DM(<address>)=Rn

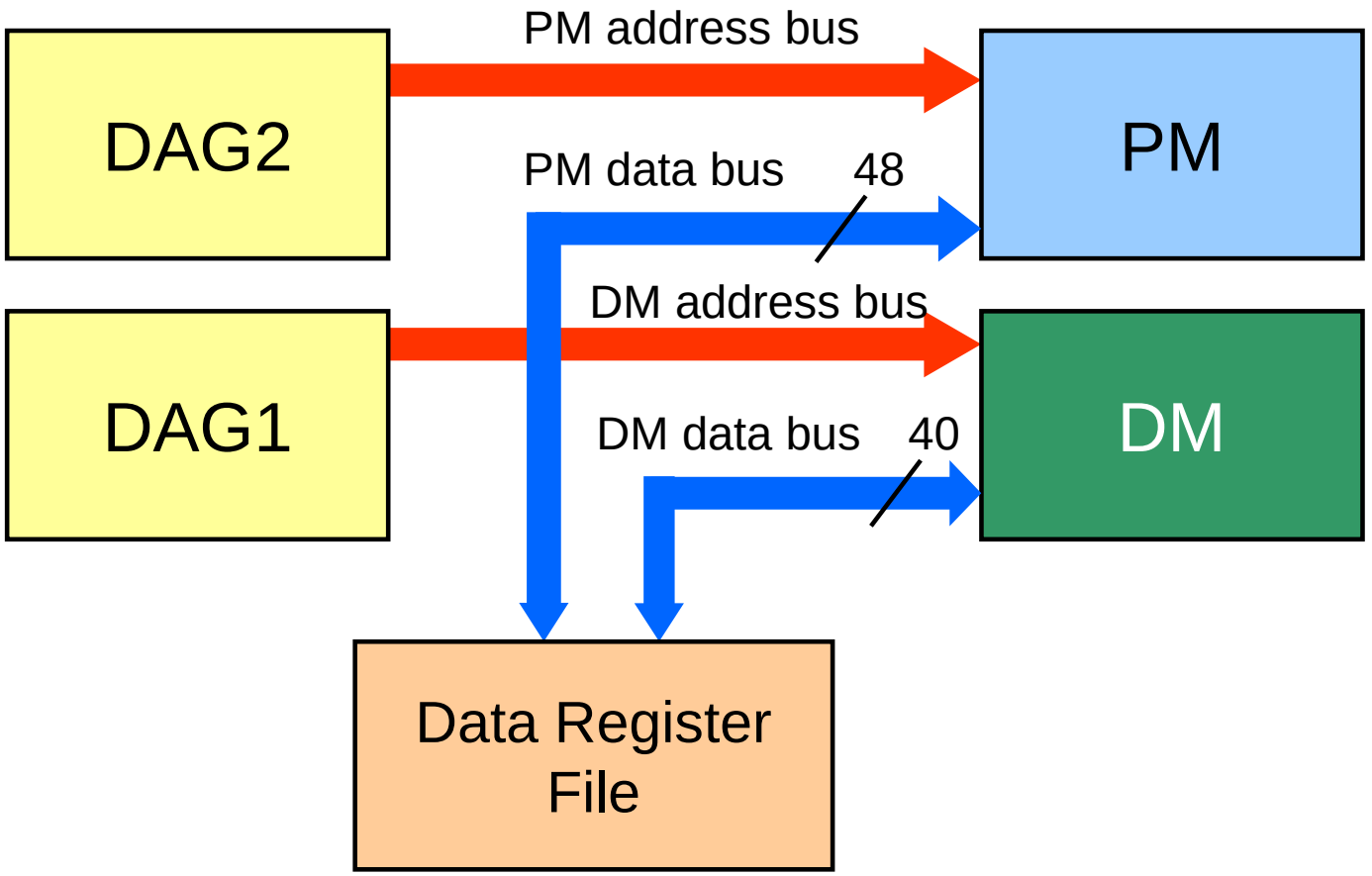
When using both memories two operands may be fetched in a single instruction.

Example:

```
R1=DM(M0,I0), R2=PM(M8,I8);
```

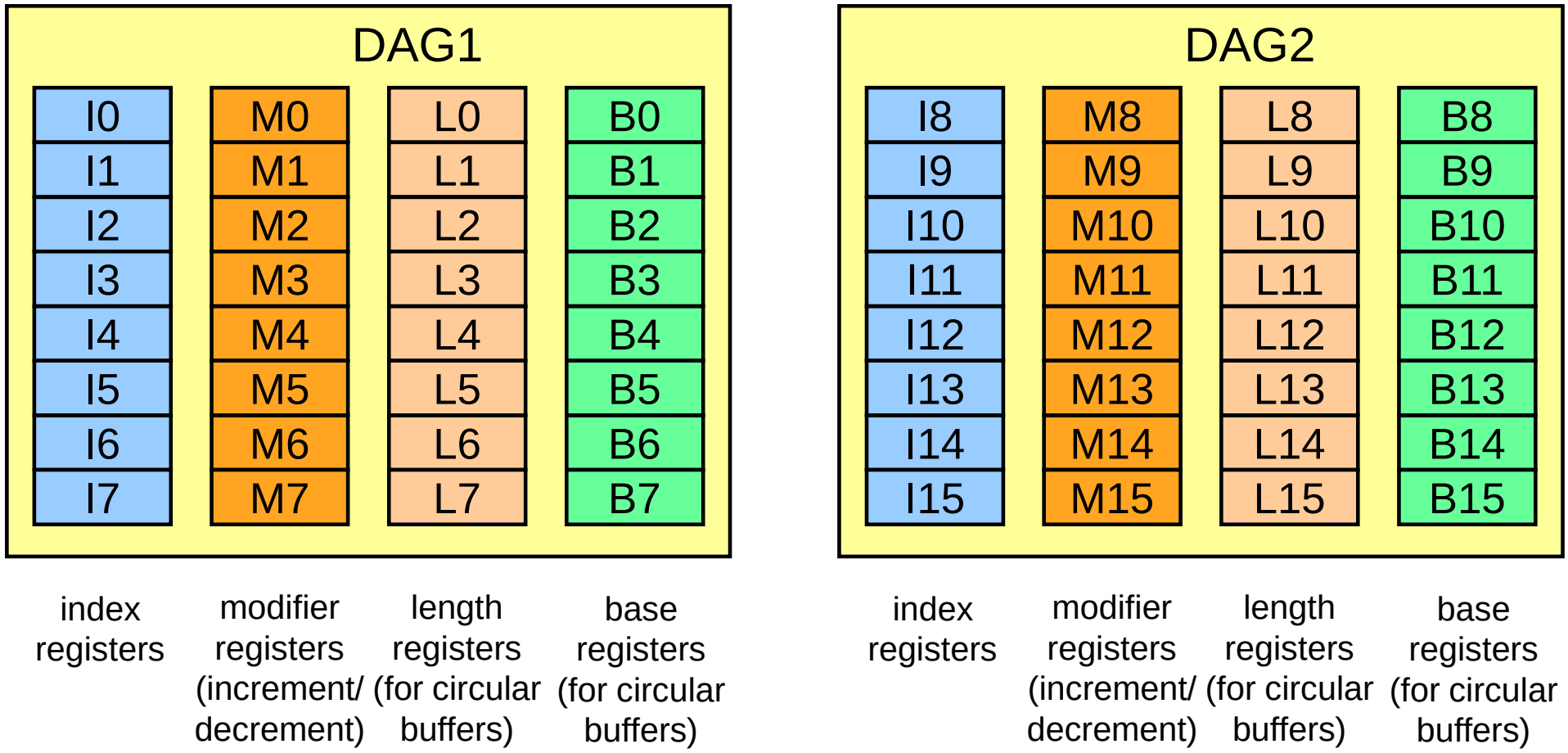
SHARC Addressing (2)

The architecture supplies special address register files to control loading and storing of data. There are two sets of such registers, called data address generators (DAGs) – one for program memory, one for data memory.



SHARC Addressing (3)

Each data address generator has eight sets of four registers:



The DAG registers can be written and read from the general register file and can be loaded with immediate values.

SHARC Addressing (4)

Addressing modes

Direct addressing

The instruction provides an immediate value as an address. Examples:

```
R0=PM(0x2000000)
R1=DM(_b)
```

Post-modify with update

This mode uses an I register and a modifier which may be an M register or an immediate value. After addressing, I is updated with the modifier. Example:

```
F0=DM(I3,M1)        ! load from address in I3 and then add M1 to I3
DM(I2,1)=R1         ! store R1 content at address in I2, then incr. I2
```

SHARC Addressing (5)

Base-plus-offset

This mode also uses an I register and a modifier which may be an M register or an immediate value. After addressing, I is *not* updated with the modifier. Examples:

```
R0=DM(M1,I3)          ! load from address=M1+I3, (no update!)
DM(1,I2)=R1           ! store R1 content at address=I2+1, (no update!)
```

Circular buffer addressing

A circular buffer is an array of *n* elements where element indexing wraps around from the end to the beginning of the buffer. The base address of the array is stored in a B register, the length *n* is stored in an L register. When an I register used in post-modify mode becomes larger than B+L, L is subtracted from I so that I "wraps around" and points back into the buffer.

SHARC Instruction Set (5)

Flow Control

Similar to ARM, there is a branch instruction called JUMP. There are three addressing modes for jumps:

- direct: the instruction specifies a 24-bit address as immediate
- indirect: the address is supplied by the DAG2 data address generator
- PC-relative: as in ARM, an immediate value is added to the current PC.

Conditional jumps

Jumps can be made conditional by adding a condition code.

```
IF GT JUMP foo;          ! jump to foo only if ALU result > 0
```

SHARC Instruction Set (6)

Condition codes

Code	Condition	Complement Code
EQ	ALU zero	NE
LT	ALU<0	GE
LE	ALU ≤ 0	GT
AC	ALU carry	NOT AC
AV	ALU overflow	NOT AV
MV	Multiplier overflow	NOT MV
MS	Multiplier Sign	NOT MS
SV	Shifter overflow	NOT SV
SZ	Shifter zero	NOT SZ
TF	Bit tested was 1	NOT TF
LCE	Loop counter expired	NOT LCE

Conditional execution

As in ARM, many operations can be made conditional.

SHARC Instruction Set (7)

Loop control

In digital signal processing, loops with counters are naturally encountered very often. SHARC provides a specialized loop instruction.

```
LCNTR=n, DO label UNTIL LCE;
```

The loop instruction specifies the length of the loop, n, the label that gives the address for the last instruction, and the loop termination condition (LCE = "loop counter expired").

When the last instruction is reached, control immediately returns to the beginning of the loop (unless the loop counter has expired). This is called *zero-overhead loop*.

The loop instruction handler uses a PC stack (30 deep) and the loop stack (6 deep).

SHARC Instruction Set (8)

Subroutine support

SHARC directly supports procedure calls with a 30-position-deep PC stack built into the CPU.

Assembler	Operation
CALL foo;	call procedure at address foo: push return address on PC stack, then set PC=foo
RTS;	return from subroutine: copy top of PC stack into PC, and pop PC stack

CALLs may use absolute, indirect or PC-relative addressing (just like JUMPs).

CALLs may be executed conditionally. Example:

```
IF GT CALL (PC,100);      ! If ALU result > 0 jump 100 locs. past current PC
```

Summary

Although the basic principles of computing are the same for all CPUs, the individual architectures can differ greatly. High-level programming languages hide most of the architectural differences from the programmer. They allow specification of the functional characteristics of the system, and abstract from non-functional characteristics such as program size and speed. These, however, can vary greatly from one CPU architecture to the other.