### Instruction sets
- Computer architecture taxonomy.
- Assembly language.

### von Neumann architecture
- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
  - Separate CPU and memory distinguishes programmable computer.
  - CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

### CPU + memory
![Diagram showing CPU and memory interaction]

### Harvard architecture
![Diagram showing Harvard architecture]

### von Neumann vs. Harvard
- Harvard can’t use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

### RISC vs. CISC
- Complex instruction set computer (CISC):
  - many addressing modes;
  - many operations.
- Reduced instruction set computer (RISC):
  - load/store;
  - pipelinable instructions.
**Instruction set characteristics**

- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

**Programming model**

- Programming model: registers visible to the programmer.
- Some registers are not visible (IR).

**Multiple implementations**

- Successful architectures have several implementations:
  - varying clock speeds;
  - different bus widths;
  - different cache sizes;
  - etc.

**Assembly language**

- One-to-one with instructions (more or less).
- Basic features:
  - One instruction per line.
  - Labels provide names for addresses (usually in first column).
  - Instructions often start in later columns.
  - Columns run to end of line.

**Pseudo-ops**

- Some assembler directives don't correspond directly to instructions:
  - Define current address.
  - Reserve storage.
  - Constants.

**ARM instruction set**

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control.
### ARM versions
- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

### ARM assembly language
- Fairly standard assembly language:
  ```assembly
  LDR r0,[r8] ; a comment
  label ADD r4,r0,r1
  ```

### ARM programming model

| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15 (PC) |
|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-------|
| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15 (PC) |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |       |
|     | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18    |
|     |    |    |    |    |    |    |    |    |    |    |    |    |    | CPSR   |

### Endianness
- Relationship between bit and byte/word ordering defines endianness:

<table>
<thead>
<tr>
<th>little-endian</th>
<th>big-endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit 31</td>
<td>bit 0</td>
</tr>
<tr>
<td>byte 3</td>
<td>byte 2</td>
</tr>
<tr>
<td>byte 1</td>
<td>byte 0</td>
</tr>
</tbody>
</table>

### ARM data types
- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
  - Address 4 starts at byte 4.
  - Can be configured at power-up as either little- or bit-endian mode.

### ARM status bits
- Every arithmetic, logical, or shifting operation sets CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  - \(-1 + 1 = 0\): NZCV = 0110.
  - \(2^{31}-1+1 = -2^{31}\): NZCV = 0101.
**ARM data instructions**

- **Basic format:**
  ```
  ADD r0,r1,r2
  ```
  Computes r1+r2, stores in r0.

- **Immediate operand:**
  ```
  ADD r0,r1,#2
  ```
  Computes r1+2, stores in r0.

**ARM data instructions**

- **ADD, ADC:** add (w. carry)
- **SUB, SBC:** subtract (w. carry)
- **RSB, RSC:** reverse subtract (w. carry)
- **MUL, MLA:** multiply (and accumulate)
- **AND, ORR, EOR**
- **BIC:** bit clear
- **LSL, LSR:** logical shift left/right
- **ASL, ASR:** arithmetic shift left/right
- **ROR:** rotate right
- **RRX:** rotate right extended with C

**Data operation varieties**

- **Logical shift:**
  - fills with zeroes.
- **Arithmetic shift:**
  - fills with ones.
- **RRX:** performs 33-bit rotate, including C bit from CPSR above sign bit.

**ARM comparison instructions**

- **CMP:** compare
- **CMN:** negated compare
- **TST:** bit-wise test
- **TEQ:** bit-wise negated test
- These instructions set only the NZCV bits of CPSR.

**ARM move instructions**

- **MOV, MVN:** move (negated)
  ```
  MOV r0, r1 ; sets r0 to r1
  ```

**ARM load/store instructions**

- **LDR, LDRH, LDRB:** load (half-word, byte)
- **STR, STRH, STRB:** store (half-word, byte)
- **Addressing modes:**
  - register indirect: LDR r0,[r1]
  - with second register: LDR r0,[r1,-r2]
  - with constant: LDR r0,[r1,#4]
ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:
  
  \[
  \text{ADR } r1,\text{FOO}
  \]

Example: C assignments

- C:
  \[
  x = (a + b) - c;
  \]
- Assembler:
  
  \[
  \begin{align*}
  &\text{ADR } r4,a; \quad \text{get address for } a \\
  &\text{LDR } r0,[r4]; \quad \text{get value of } a \\
  &\text{ADR } r4,b; \quad \text{get address for } b, \text{reusing } r4 \\
  &\text{LDR } r1,[r4]; \quad \text{get value of } b \\
  &\text{ADD } r3,r0,r1; \quad \text{compute } a \cdot b \\
  &\text{ADR } r4,c; \quad \text{get address for } c \\
  &\text{LDR } r2,[r4]; \quad \text{get value of } c
  \end{align*}
  \]

C assignment, cont’d.

- C:
  \[
  y = a \cdot (b+c);
  \]
- Assembler:
  
  \[
  \begin{align*}
  &\text{ADR } r4,b; \quad \text{get address for } b \\
  &\text{LDR } r0,[r4]; \quad \text{get value of } b \\
  &\text{ADR } r4,c; \quad \text{get address for } c \\
  &\text{LDR } r1,[r4]; \quad \text{get value of } c \\
  &\text{ADD } r2,r0,r1; \quad \text{compute partial result} \\
  &\text{ADR } r4,a; \quad \text{get address for } a \\
  &\text{LDR } r0,[r4]; \quad \text{get value of } a \\
  &\text{MUL } r2,r2,r0; \quad \text{compute final value for } y \\
  &\text{ADR } r4,y; \quad \text{get address for } y \\
  &\text{STR } r2,[r4]; \quad \text{store } y
  \end{align*}
  \]

Example: C assignment

- C:
  \[
  z = (a \ll 2) \mid (b \& 15);
  \]
- Assembler:
  
  \[
  \begin{align*}
  &\text{ADR } r4,a; \quad \text{get address for } a \\
  &\text{LDR } r0,[r4]; \quad \text{get value of } a \\
  &\text{MOV } r0,16; \quad \text{shift } \ll 2 \\
  &\text{ADR } r4,b; \quad \text{get address for } b \\
  &\text{LDR } r1,[r4]; \quad \text{get value of } b \\
  &\text{ADD } r1,r1,#15; \quad \text{perform AND} \\
  &\text{LDR } r0,[r4]; \quad \text{get value of } a \\
  &\text{AND } r1,r1,#15; \quad \text{perform OR} \\
  &\text{STR } r2,[r4]; \quad \text{store } y
  \end{align*}
  \]

C assignment, cont’d.
C assignment, cont’d.

ADR r4,z ; get address for z
STR r1,[r4] ; store value for z

ARM flow of control

All operations can be performed conditionally, testing CPSR:
- EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:
  B #100
  Can be performed conditionally.

Additional addressing modes

- Base-plus-offset addressing:
  LDR r0,[r1,#16]
  Loads from location r1+16
- Auto-indexing increments base register:
  LDR r0,[r1,#16]!
- Post-indexing fetches, then does offset:
  LDR r0,[r1],#16
  Loads r0 from r1, then adds 16 to r1.

Example: if statement

C:
if (a < b) { x = 5; y = c + d; } else x = c - d;

Assembler:
; compute and test condition
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
ADR r4,b ; get address for b
LDR r1,[r4] ; get value for b
CMP r0,r1 ; compare a < b
BGE fblock ; if a >= b, branch to false block

If statement, cont’d.

; true block
MOV r0,#5 ; generate value for x
ADR r4,x ; get address for x
STR r0,[r4] ; store x
ADR r4,c ; get address for c
LDR r0,[r4] ; get value of c
ADR r4,d ; get address for d
LDR r1,[r4] ; get value of d
ADD r0,r0,r1 ; compute y
ADR r4,y ; get address for y
STR r0,[r4] ; store y
B after ; branch around false block

; false block
fblock ADR r4,c ; get address for c
LDR r0,[r4] ; get value of c
ADR r4,d ; get address for d
LDR r1,[r4] ; get value for d
SUB r0,r0,r1 ; compute c-d
ADR r4,x ; get address for x
STR r0,[r4] ; store value of x
after ...

If statement, cont’d.
Example: Conditional instruction implementation

; true block
MOVLT r0,#5 ; generate value for x
ADRLT r4,x ; get address for x
STRLT r0,[r4] ; store x
ADRLT r4,c ; get address for c
LDLT r0,[r4] ; get value of c
ADRLT r4,d ; get address for d
LDLT r1,[r4] ; get value of d
ADDLT r0,r0,r1 ; compute y
ADRLT r4,y ; get address for y
STRLT r0,[r4] ; store y

; false block
ADRLT r4,c ; get address for c
LDRGE r0,[r4] ; get value of c
ADRLT r4,d ; get address for d
LDRGE r1,[r4] ; get value of d
SUBGE r0,r0,r1 ; compute a-b
ADRLT r4,x ; get address for x
STRLT r0,[r4] ; store value of x

Example: switch statement

C:
switch (test) { case 0: ... break; case 1: ... }

Assembler:
ADR r2,test ; get address for test
LDR r0,[r2] ; load value for test
ADR r1,switchtab ; load address for switch table
LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
DCD case1
...

Example: FIR filter

C:
for (i=0, f=0; i<N; i++)
f = f + c[i]*x[i];

Assembler
; loop initiation code
MOV r0,#0 ; use r0 for i
MOV r8,#0 ; use separate index for arrays
ADR r2,N ; get address for N
LDR r1,[r2] ; get value of N
MOV r2,#0 ; use r2 for f

; loop body
LDR r4,[r3,r8] ; get c[i]
LDR r6,[r5,r8] ; get x[i]
MUL r4,r4,r6 ; compute c[i]*x[i]
ADD r2,r2,r4 ; add into running sum
ADD r0,r0,r8 ; add one word offset to array index
ADD r0,r0,1 ; add 1 to i
CMP r0,r1 ; exit?
BLT loop ; if i < N, continue

FIR filter, cont’d

ARM subroutine linkage

Branch and link instruction:
BL foo
Copies current PC to r14.
To return from subroutine:
MOV r15,r14
Summary

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
  - Some multi-register operations take longer.
- All instructions can be executed conditionally.

SHARC instruction set

- SHARC programming model.
- SHARC assembly language.
- SHARC memory organization.
- SHARC data operations.
- SHARC flow of control.

SHARC programming model

- Register files:
  - R0-R15 (aliased as F0-F15 for floating point)
  - Status registers.
  - Loop registers.
  - Data address generator registers.
  - Interrupt registers.

SHARC assembly language

- Algebraic notation terminated by semicolon:
  \[ \text{R1=DM(M0,I0), R2=PM(M8,18); ! comment} \]
  \[ \text{label: R3=R1+R2;} \]

  data memory access program memory access

SHARC data types

- 32-bit IEEE single-precision floating-point.
- 40-bit IEEE extended-precision floating-point.
- 32-bit integers.
- Memory organized internally as 32-bit words.

SHARC microarchitecture

- Modified Harvard architecture.
  - Program memory can be used to store some data.
- Register file connects to:
  - multiplier
  - shifter
  - ALU.
**SHARC mode registers**

- Most important:
  - ASTAT: arithmetic status.
  - STKY: sticky.
  - MODE 1: mode 1.

**Rounding and saturation**

- Floating-point can be:
  - rounded toward zero;
  - rounded toward nearest.
- ALU supports saturation arithmetic (ALUSAT bit in MODE1).
  - Overflow results in max value, not rollover.

**Multiplier**

Fixed-point operations can accumulate into local MR registers or be written to register file. Fixed-point result is 80 bits. Floating-point results always go to register file.

Status bits: negative, under/overflow, invalid, fixed-point undeflow, floating-point underflow, floating-point invalid.

**ALU/shifter status flags**

ALU:
- zero, overflow, negative, fixed-point carry, inputsign, floating-point invalid, last op was floating-point, compare accumulation registers, floating-point under/overflow, fixed-point overflow, floating-point invalid

Shifter:
- zero, overflow, sign

**Flag operations**

- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.
- STKY is sticky version of some ASTAT bits.

**Example: data operations**

- Fixed-point -1 + 1 = 0:
  - AZ = 1, AU = 0, AN = 0, AV = 0, AC = 1, AI = 0.
  - STKY bit AOS (fixed point underflow) not set.
- Fixed-point -2*3:
  - MN = 1, MV = 0, MU = 1, MI = 0.
  - Four STKY bits, none of them set.
- LSHIFT 0x7fffffff BY 3: SZ=0,SV=1,SS=0.
**Multifunction computations**

Can issue some computations in parallel:
- dual add-subtract;
- fixed-point multiply/accumulate and add, subtract, average
- floating-point multiply and ALU operation
- multiplication and dual add/subtract

Multiplier operand from R0-R7, ALU operand from R8-R15.

**SHARC load/store**

- Load/store architecture: no memory-direct operations.
- Two data address generators (DAGs):
  - program memory;
  - data memory.
- Must set up DAG registers to control loads/stores.

**DAG1 registers**

<table>
<thead>
<tr>
<th>10</th>
<th>M0</th>
<th>L0</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>M1</td>
<td>L1</td>
<td>B1</td>
</tr>
<tr>
<td>12</td>
<td>M2</td>
<td>L2</td>
<td>B2</td>
</tr>
<tr>
<td>13</td>
<td>M3</td>
<td>L3</td>
<td>B3</td>
</tr>
<tr>
<td>14</td>
<td>M4</td>
<td>L4</td>
<td>B4</td>
</tr>
<tr>
<td>15</td>
<td>M5</td>
<td>L5</td>
<td>B5</td>
</tr>
<tr>
<td>16</td>
<td>M6</td>
<td>L6</td>
<td>B6</td>
</tr>
<tr>
<td>17</td>
<td>M7</td>
<td>L7</td>
<td>B7</td>
</tr>
</tbody>
</table>

**Data address generators**

Provide indexed, modulo, bit-reverse indexing.
MODE1 bits determine whether primary or alternate registers are active.

**BASIC addressing**

- Immediate value:
  \[ R0 = DM(0x20000000); \]
- Direct load:
  \[ R0 = DM(_a); \] ! Loads contents of _a
- Direct store:
  \[ DM(_a) = R0; \] ! Stores R0 at _a

**Post-modify with update**

- I register holds base address.
- M register/immediate holds modifier value.
  \[ R0 = DM(I3,M3) ! Load \]
  \[ DM(I2,1) = R1 ! Store \]
- Circular buffer: L register is buffer start index, B is buffer base address.
Data in program memory

- Can put data in program memory to read two values per cycle:
  \[ F_0 = DM(M_0, I_0), F_1 = PM(M_8, I_9); \]
- Compiler allows programmer to control which memory values are stored in.

Example: C assignments

- C:
  \[ x = (a + b) - c; \]
- Assembler:
  
  ```asm
  R0 = DM(_a); ! Load a
  R1 = DM(_b); ! Load b
  R3 = R0 + R1;
  R2 = DM(_c); ! Load c
  R3 = R3-R2;
  DM(_x) = R3; ! Store result in x
  ```

Example, cont’d.

- C:
  \[ y = a \ast (b+c); \]
- Assembler:
  
  ```asm
  R1 = DM(_b); ! Load b
  R2 = DM(_c); ! Load c
  R2 = R1 + R2;
  R0 = DM(_a); ! Load a
  R2 = R2*R0;
  DM(_y) = R2; ! Store result in y
  ```

Example, cont’d.

- Shorter version using pointers:
  
  ```asm
  ! Load b, c
  R2=DM(I1,M5), R1=PM(I8,M13);
  R0 = R2+R1, R12=DM(I0,M5);
  R6 = R12*R0;
  DM(I0,M5)=R8; ! Store in y
  ```

Example, cont’d.

- C:
  \[ z = (a << 2) | (b \& 15); \]
- Assembler:
  
  ```asm
  R0=DM(_a); ! Load a
  R0=LSHIFT R0 by #2; ! Left shift
  R1=DM(_b); R3=#15; ! Load immediate
  R1=R1 AND R3;
  R5 = R1 OR R0;
  DM(_z) = R0;
  ```

SHARC program sequencer

- Features:
  - instruction cache;
  - PC stack;
  - status registers;
  - loop logic;
  - data address generator;
### Conditional instructions

Instructions may be executed conditionally. Conditions come from:
- arithmetic status (ASTAT);
- mode control 1 (MODE1);
- flag inputs;
- loop counter.

### SHARC jump

- Unconditional flow of control change:
  - JUMP foo
- Three addressing modes:
  - direct;
  - indirect;
  - PC-relative.

### Branches

Types: CALL, JUMP, RTS, RTI. Can be conditional.
Address can be direct, indirect, PC-relative. Can be delayed or non-delayed.
JUMP causes automatic loop abort.

### Example: C if statement

- C:
  ```c
  if (a < b) { x = 5; y = c + d; } 
  else x = c - d;
  ```
- **Assembler**:
  ```assembly
  ! Test 
  R0 = DM(_a); R1 = DM(_b); 
  COMP(R0,R1); ! Compare IF GE JUMP fblock;
  ```

### Example: C if statement, cont’d.

- ! True block
  ```assembly
  tblock: R0 = 5; ! Get value for x 
  DM(_x) = R0; 
  R0 = DM(_c); R1 = DM(_d); 
  R1 = R0+R1; 
  DM(_y)=R1; 
  JUMP other; ! Skip false block
  ```

- ! False block
  ```assembly
  fblock: R0 = DM(_c); 
  R1 = DM(_d); 
  R1 = R0-R1; 
  DM(_x) = R1; 
  other: ! Code after if
  ```
Fancy if implementation

- **C:**
  
  ```c
  if (a>b) y = c-d; else y = c+d;
  ```

  Use parallelism to speed it up---compute both cases, then choose which one to store.

Fancy if implementation, cont’d.

- ! Load values
  
  ```
  R1=DM(_a); R8=DM(_b);
  R2=DM(_c); R4=DM(_d);
  ```

- ! Compute both sum and difference
  
  ```
  R12 = r2+r4, r0 = r2-r4;
  ```

- ! Choose which one to save
  
  ```
  comp(r8,r1);
  ```

- if ge r0=r12;

- ! Write to y
  
  ```
  dm(_y) = r0
  ```

DO UNTIL loops

DO UNTIL instruction provides efficient looping:

- ! setup
  
  ```
  I0=DM(I0,M0), I8=PM(I8,M8);
  ```

- ! Loop body
  
  ```
  LCNTR=30, DO label UNTIL LCE;
  ```

Example: FIR filter

- **C:**
  
  ```c
  for (i=0, f=0; i<N; i++)
  ```

  ```c
  f = f + c[i]*x[i];
  ```

FIR filter assembler

- ! setup
  
  ```
  I0=DM(I0,M0), I8=PM(I8,M8);
  ```

- ! Loop body
  
  ```
  LCNTR=30, DO loopend UNTIL LCE;
  ```

SHARC subroutine calls

- ! Use CALL instruction:
  
  ```
  CALL foo;
  ```

- Can use absolute, indirect, PC-relative addressing modes.

- Return using RTS instruction.
**PC stack**

PC stack: 30 locations X 24 instructions. Return addresses for subroutines, interrupt service routines, loops held in PC stack.

---

**Example: C function**

- **C:**
  ```c
  void f1(int a) { f2(a); }
  ```

- **Assembler:**
  ```
  f1: R0=DM(I1,-1); ! Load arg into R0
  DM(I1,ML)=R0; ! Push f2's arg
  CALL f2;
  MODIFY(I1,-1); ! Pop element
  RTS;
  ```