嵌入式微處理機系統

國立台灣科技大學

電機系

王乃堅老師

Feb. 2024

03_ARM Instruction Sets

Instruction sets

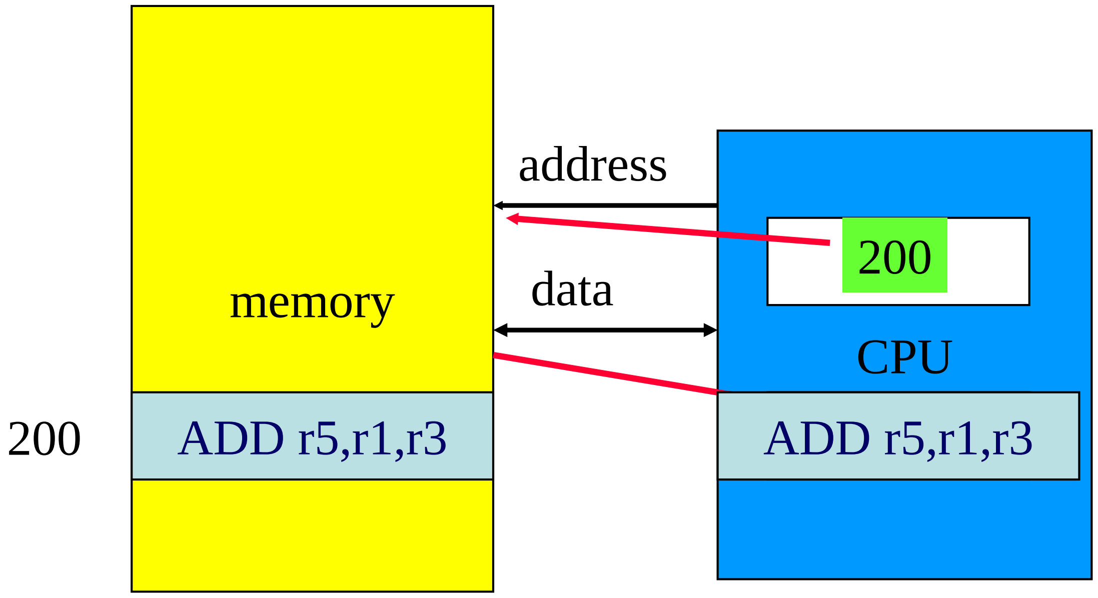
- Computer architecture taxonomy.
- Assembly language.

von Neumann architecture

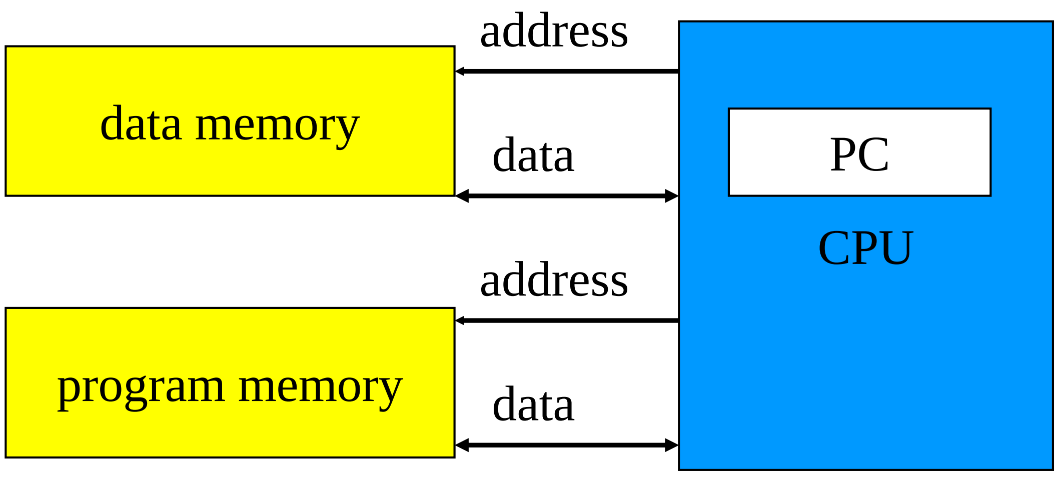
- Memory holds data + instructions
- CPU fetches instructions from memory, decodes instruction, and executes it.
  - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out:
  - program counter (**PC**): point to an instruction in memory
  - instruction register (**IR**),
  - general-purpose registers, etc.

von-Neumann architecture

- CPU + memory



Harvard architecture



Harvard architecture

- Separate memories for data and program
- Program Counter (PC): point to program memory not data
- Harder to write self-modifying programs (programs that write data values, then use those values as instructions)
- Widely used today for one simple reason – the separation of program and data memories provides higher performance for DSP
- Higher memory bandwidth

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.

ARM assembly language example

- r0 $\leftarrow$ c-d

```
label1      ADR r4,c
            LDR r0,[r4]      ;  r0  ← [r4] =c
            ADR r4,d
            LDR r1,[r4]      ;  r1  ← [r4] =d
            SUB r0,r0,r1      ; r0 ← r0-r1=c-d
```

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.

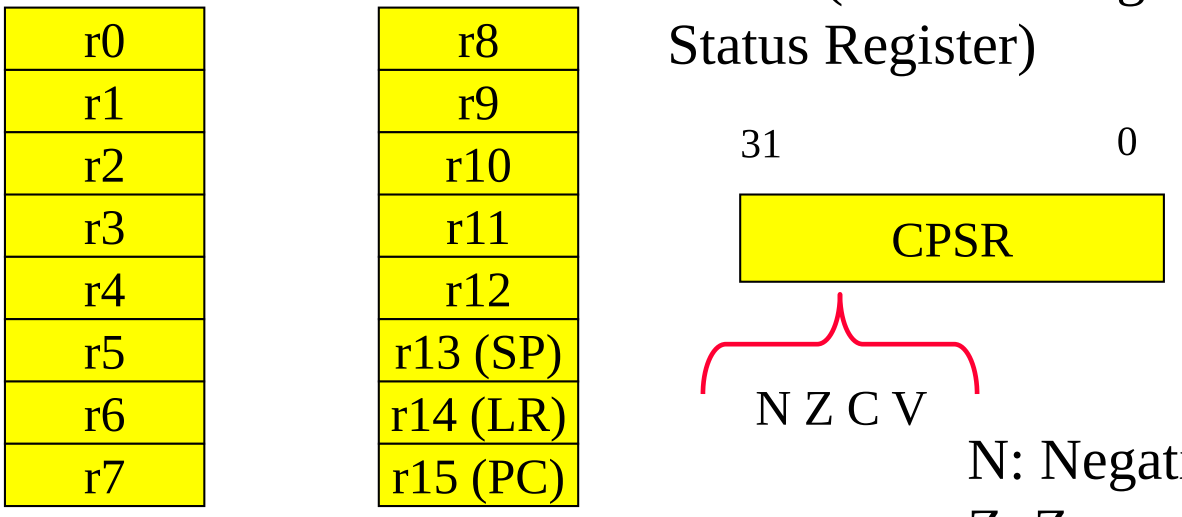
ARM assembly language

- Fairly standard assembly language:

```
        LDR r0,[r8]       ; a comment
label   ADD r4,r0,r1
```

ARM programming model

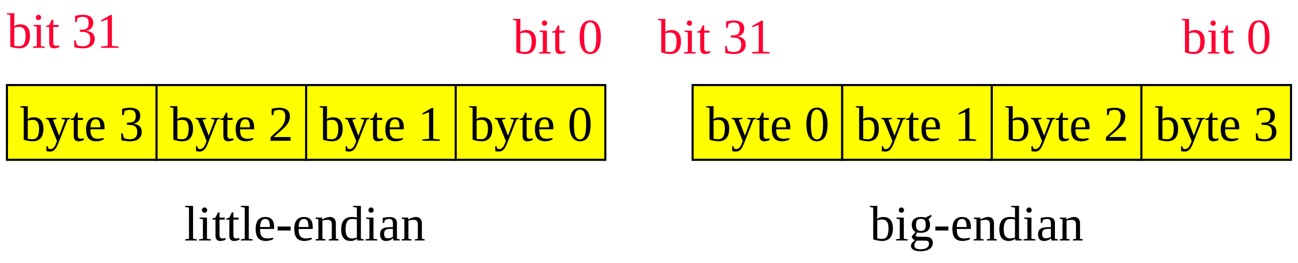
CPSR (Current Program Status Register)



N: Negative
Z: Zero
C: Carry
V: Overflow

Endianness

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



- Ex. Store 12345678H in the memory location start at 10000H.

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 big-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.
    - 0xffffffff+0x1=0x0
  - 0-1=-1: NZCV = 1000.
    - 0x0-0x1=0xffffffff
  - 2^{31}-1+1 = 2^{31}: NZCV = 1001.
    - 0x7fffffff+0x1=0x80000000

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.
    - 0xffffffff+0x1=0x0
  - 0-1=-1: NZCV = 1000.
    - 0x0-0x1=0xffffffff
  - 2^{31}-1+1 = 2^{31}: NZCV = 1001.
    - 0x7fffffff+0x1=0x80000000

ARM data instructions

- Basic format:

  ADD r0,r1,r2 ; r0 $\leftarrow$ r1 + r2,
- Immediate operand:

  ADD r0,r1,#2 ; r0 $\leftarrow$ r1 + 2,

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 test only the NZCV bits of CPSR.

ARM instructions

- MOV, MVN : move (negated)

  MOV r0, r1 ; r0 $\leftarrow$ r1
- SUB, RSB : (reverse) subtract

  SUB r0, r1,r2 ; r0 $\leftarrow$ r1 - r2

  RSB r0, r1,r2 ; r0 $\leftarrow$ r2 - r1
- BIC : bit clear

  BIC r0, r1,r2 ; r0 $\leftarrow$ r1(with r2 mask)

  ; r0 $\leftarrow$ r1•(~r2)

  ; if r1=FFH, r2=0FH $\rightarrow$r0=F0H

ARM instructions

- MUL, MLA : multiply (and accumulate)
  - MUL multiplies two values, but with some restrictions:
    - No operand may be an immediate
    - Two source operands must be different registers
  - MLA performs a multiply-accumulate operation, particularly useful in matrix operation and signal processing.
    - MLA r0,r1,r2,r3 ; r0 $\leftarrow$ r1xr2+r3

ARM ADS download

- http://www.arm.com/support/downloads/info/3771.html

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:

  ADR r1,FOO

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a;  r0 ← a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b;  r1 ← b
ADD r3,r0,r1      ; compute a+b;  r3 ← a+b
ADR r4,c          ; get address for c
LDR r2,[r4]       ; get value of c;  r2 ← c
SUB r3,r3,r2      ; complete computation of x;  r3 ← a+b-c
ADR r4,x          ; get address for x
STR r3,[r4]       ; store value of x , r3 →[r4]
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b          ; get address for b
LDR r0,[r4]       ; get value of b
ADR r4,c          ; get address for c
LDR r1,[r4]       ; get value of c
ADD r2,r0,r1      ; compute partial result
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
MUL r2,r2,r0      ; compute final value for y
ADR r4,y          ; get address for y
STR r2,[r4]       ; store y
```

Example: C assignment

- C:

```
z = (a << 2) |  (b & 15);
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
MOV r0,r0,LSL 2   ; perform shift
ADR r4,b          ; get address for b
LDR r1,[r4]       ; get value of b
AND r1,r1,#15     ; perform AND
ORR r1,r0,r1      ; perform OR
ADR r4,z          ; get address for z
STR r1,[r4]       ; store value for z
```

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.

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.

Example: if statement (1/3)

- 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 (2/3)

```
; 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
```

If statement (3/3)

```
; 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 ...
```

Example: Conditional instruction implementation (1/2)

```
; 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
    LDRLT r0,[r4]      ; get value of c
    ADRLT r4,d         ; get address for d
    LDRLT r1,[r4]      ; get value of d
    ADDLT r0,r0,r1      ; compute y
    ADRLT r4,y          ; get address for y
    STRLT r0,[r4]       ; store y
```

Conditional instruction implementation (2/2)

```
; false block
    ADRGE r4,c           ; get address for c
    LDRGE r0,[r4]         ; get value of c
    ADRGE r4,d           ; get address for d
    LDRGE r1,[r4]         ; get value for d
    SUBGE r0,r0,r1        ; compute a-b
    ADRGE r4,x            ; get address for x
    STRGE 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 r15,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
          DCD case1
          ...
case0     …                    ; code for case0
case1     …                    ; code for case1
```

Example: FIR filter (1/2)

- 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
```

FIR filter (2/2)

```
        ADR r3,c            ; load r3 with base of c
        ADR r5,x            ; load r5 with base of x
; loop body
loop 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 r8,r8,#4        ; 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
```

ARM subroutine linkage

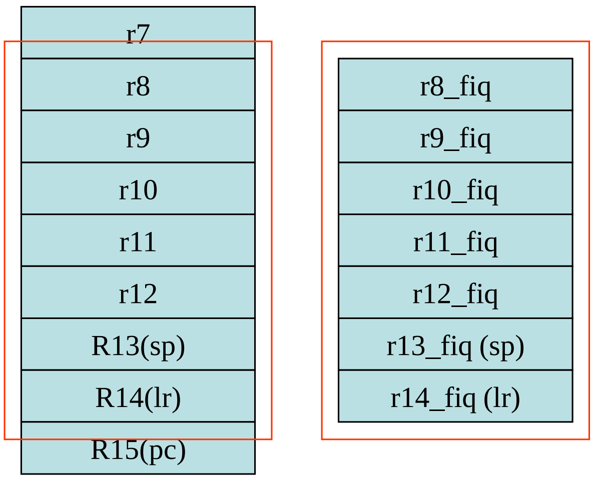
- Branch and link instruction:

  BL foo

  – Copies current PC to r14.

- To return from subroutine:

  MOV r15,r14



Nested subroutine calls

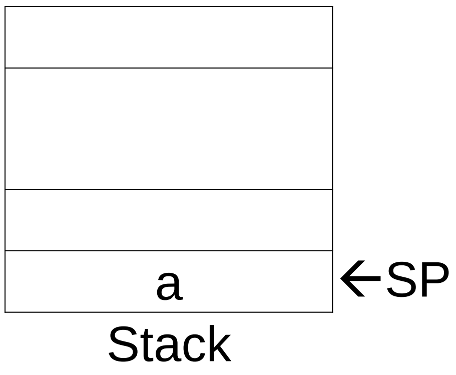
- Nesting/recursion requires coding convention:

```
f1    LDR r0,[r13]!          ; load arg into r0 from stack

      ; f1() call f2()
      STR r0,  [r13]!        ; store arg to f2 on stack
      BL f2                  ; branch and link to f2
      STR r14,[r13]!         ; store f1's return adrs

      ; return from f2() to f1()
      SUB r13,#4             ; pop f2's arg off stack
      LDR r15,[r13]!         ; restore register and return
```

C code

```
void f2( int x)  {
   /* do sth…*/
}
void f1( int a)  {
   f2(a);
}
```

Stack operation(Animation)



Assembly

Summary

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