ARM instruction set

Used in PDA's, telephones, video games, ...

- ARM assembly language.
- ARM programming model.
- ARM data operations.
- ARM flow of control.
- Assembly vs C

ARM versions

- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.
- RISC, von Neumann architecture

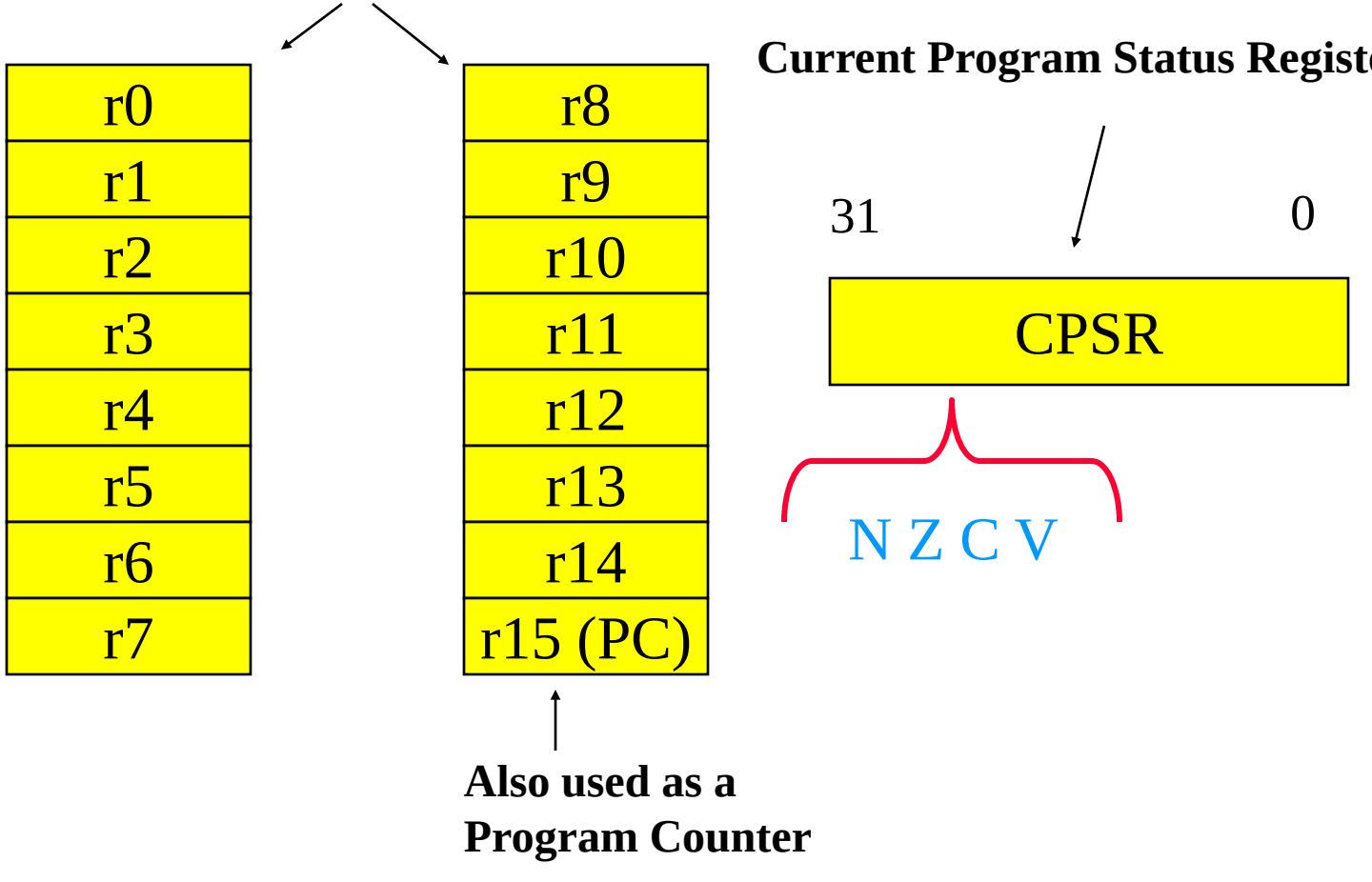
ARM assembly language

- Fairly standard assembly language:

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

ARM programming model

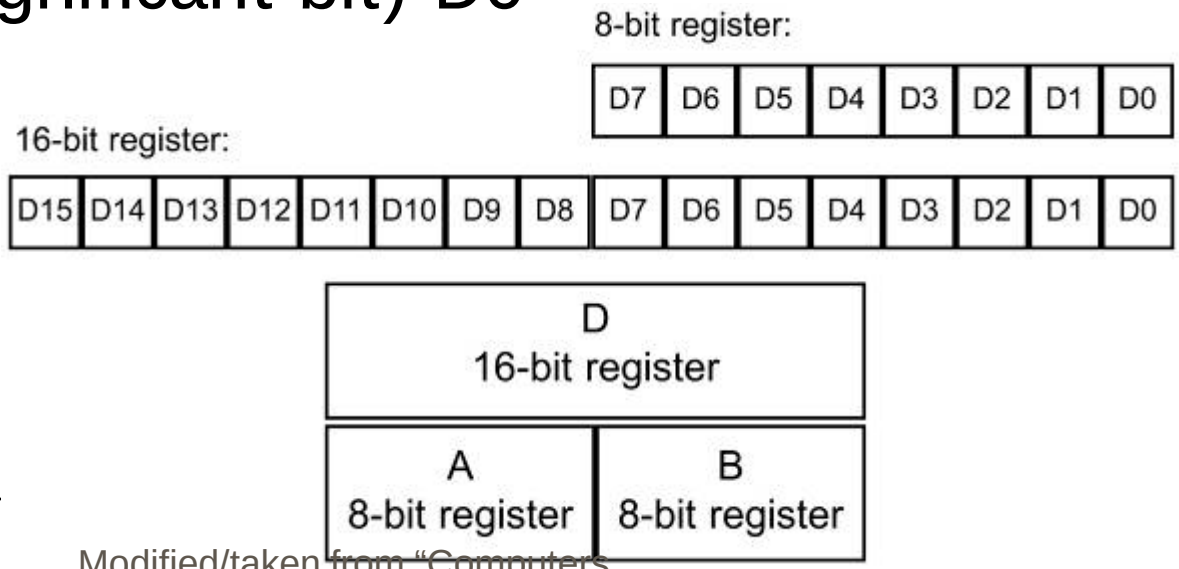
Sixteen 32 bit registers



Modified/taken from “Computers as Components, by W.Wolf”

Recap: HCS12 Registers

- **Registers -** store information temporarily.
  - HCS12 registers are either 8-bit or 16-bit
- The 8 bits of a register are shown here.
  - MSB (most significant bit) D7 to LSB (least significant bit) D0



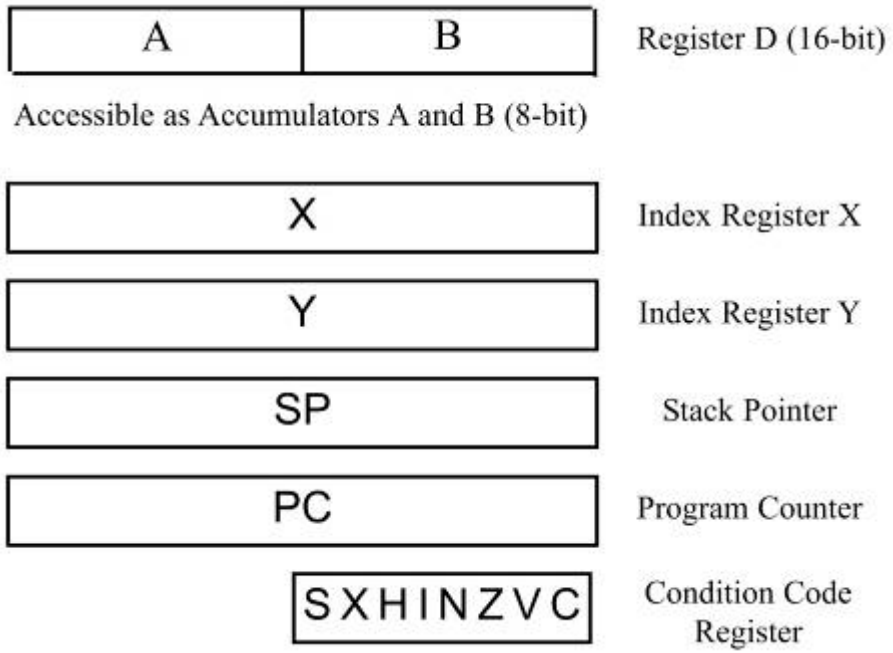
Modified/taken from "Computers as Components, by W.Wolf"

HCS12 Registers

- The most widely used registers are:

A, B, X, Y, SP (stack pointer), CCR (condition code register), and PC (program counter).

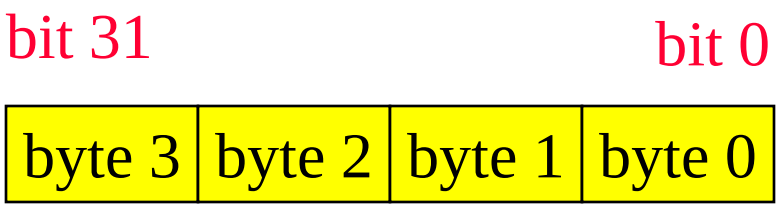
- All 16-bits except A, B & CCR.
- A & B registers are referred to as accumulators.
  - combining A & B gives 16-bit register D



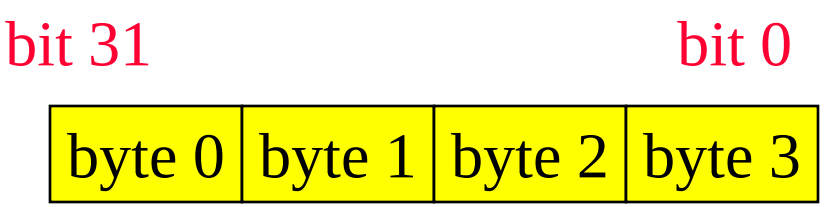
Modified/taken from "Computers as Components, by W.Wolf"

Endianness

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



little-endian (little-end comes first)



big-endian (big-end comes first)

- J. Swift's Gulliver's Travels: **L-E & B-E differed in the way they ate a hard-boiled egg (from small end vs big end)!**

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: word 0 → loc 0, word 1→loc 4, word 2→loc 8, ...
- 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: 0xffffffff+0x1=0x0 → NZCV = 0110.**
  - **0-1=-1: 0x0 – 0x1 =0xffffffff → NZCV=1000.**
  - **2^{31}-1+1 = -2^{31}: 0x7fffffff+0x1=0x80000000 → NZCV = 1001 (overflow: positive's add up to negative)**

ARM data instructions

- Basic format:

```
ADD r0,r1,r2
```

  - Computes r1+r2, stores in r0, i.e., **r0=r1+r2**.

- Immediate operand:

```
ADD r0,r1,#2
```

  - Computes r1+2, stores in r0: **r0=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), e.g.,

MLA r0,r1,r2,r3 r0=r1*r2+r3

- AND, ORR, EOR (exclusive OR)
- BIC: bit clear→ BIC r0, r1,r2 (set r0 to r1 according to mask r2)
- LSL, LSR: logical shift left/right
- ASL, ASR: arithmetic shift left/right → copies sign bit
- ROR: rotate right
- RRX : rotate right extended-with C (carry)

Matrix, DSP, …

Modified/taken from “Computers as Components, by W.Wolf”

Data operation varieties

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

ARM comparison instructions

- CMP: compare → **CMP r0,r computes r0 - r1 and sets NZCV**
- CMN: negated compare → **CMN r0,r1 → r0+r1, set status bits**
- TST: bit-wise test (AND) on operands
- TEQ: bit-wise negated test (XOR) on operands → op1 exclusive or with op2
- These instructions set only the NZCV bits of CPSR.

ARM move instructions

- MOV, MVN : move (negated, i.e., ones-complement)

```
MOV r0, r1 ; sets r0 to r1
```

ARM load/store instructions: from/to memory

- LDR, LDRH, LDRB: load (half-word, byte)
- STR, STRH, STRB: store (half-word, byte)
- Addressing modes (ARM address is 32bits)
  - **register indirect : `LDR r0,[r1], STR r0,[r1]`**
  - **with second register : `LDR r0,[r1,-r2]`→ `loads r0 from [r1-r2]`**
  - **with constant : `LDR r0,[r1,#4]` → `from r1+4 address`**

ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Can generate value by performing arithmetic on PC, **i.e., SUB r1,r15,#101 (distance to desired address)**
- ADR pseudo-op generates instruction required to calculate address:

```
ADR r1,FOO → to get an address FOO
  (e.g., 0x100) into r1.
```

Example: C assignments

- C:

```
x = (a + b) - c; //r0:a, r1:b, r2:c, r3:x, r4: reusable
```

- Assembler:

```
ADR r4,a        ; get address for a
LDR r0,[r4]     ; get value of a
ADR r4,b        ; get address for b, reusing r4
LDR r1,[r4]     ; get value of b
ADD r3,r0,r1    ; compute a+b
ADR r4,c        ; get address for c
LDR r2,[r4]     ; get value of c
```

C assignment, cont'd.

```
SUB r3,r3,r2    ; complete computation of x (r3=
                ; r3-r2)
ADR r4,x        ; get address for x
STR r3, [r4]    ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c); //r0: a,b, r1:c, r2:y, r4: addresses
```

- 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 b+c
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.

```
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: bitwise OR, AND

```
z = (a << 2) |  (b & 15); //r0:a,z, r1:b, r4:
                                  addresses
```

- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift, i.e.,  a<<2
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
```

C assignment, cont'd.

```
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]` → Offset up to 4096 (2^14)

  - Loads from location r1+16
- Auto-indexing increments base register:

  `LDR r0,[r1,#16]!` →`r1 is updated with [r1+16]`
- Post-indexing fetches, then does offset:

  `LDR r0,[r1],#16`

  - Loads r0 from r1, then adds 16 to r1.
  - **Last 2 examples:** Same final values for r1 but different for r0

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  - EQ, NE, CS (carry set), CC, MI (minus) , PL (plus), VS (overflow), VC (no overflow) , HI, LS, GE, LT, GT, LE
    - → e.g., ADDEQ, ADDGT, SUBLT, ...
- Branch operation:

  B #100 → `# of words to jump (adds 400 to the current PC value)`

  - Can be performed conditionally.

Example: if statement

- C:

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

- Assembler:

```
; compute and test condition:  (a>b) ?= TRUE
  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
```

Modified/taken from “Computers as Components, by W. Wolf”

If statement, cont'd.

```
; true block → x=5, y=c+d
  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, cont'd.

```
; false block → x=c-d
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 a-b
       ADR r4,x ; get address for x
       STR r0,[r4] ; store value of x
after ...
```

Example: FIR filter $f=\Sigma Ci\ Xi$

- 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 as index for arrays: c x
  ADR r2,N ; get address for N
  LDR r1,[r2] ; get value of N
  MOV r2,#0 ; use r2 for f
```

FIR filter, cont'.d

```
   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 (4bytes)to
                      array index
      ADD r0,r0,#1   ; add 1 to i
      CMP r0,r1      ; exit?
      BLT loop       ; if i < N, continue
```

Modified/taken from "Computers as Components, by W.Wolf"

ARM subroutine linkage

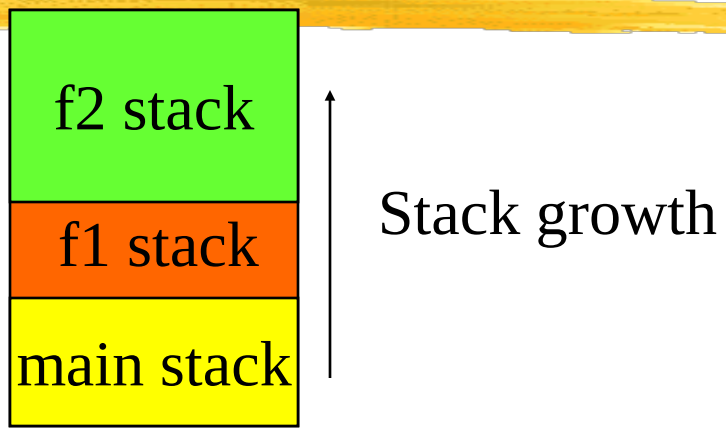
C Function calls:

```
x=a+b;
foo(x);
y=c-d
```

- **Branch and link instruction:**
  - **`BL foo`** → `link to code starting at location "foo"`
  - **Copies current PC to r14.**
- **To return from subroutine: (must restore PC)**
  - **MOV r15, r14** → **r14 should not be overwritten during "foo"**

Nested subroutines

```
main( ){
        int x=2, y;
        … C code
        f1(x);
        … more C code
}
void f1(int a){
        f2(a);
}
void f2(int r){
…}
```



➔ **Parameters, return addresses, CPU registers → pushed onto the stack before call to a procedure**

➔ **Values are popped off on return from subroutine**

Summary

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
  - Some multi-register operations take longer.
- C and Assembly equivalents

- ASSIGNEMENT #2: Q2-9 & Q2-10(a)