بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

CSE211s:

Introduction to Embedded Systems

ARM Assembly Language

﴿يَرْفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُوا۟ مِنكُمْ وَٱلَّذِينَ أُوتُوا۟ ٱلْعِلْمَ دَرَجَٰتٍ﴾

* Assembly language:

→ Assembly language follows this relatively structured form to make it easy for the assembler to read the program & to consider most aspects of the program line by line

```
<label>      opcode  operands    ;comment
```

→ When describing assembly instructions we will use the following list of symbols

Ra Rd Rm Rn Rt and **Rt2** represent registers
{Rd,} represents an optional destination register
#imm12 represents a 12-bit constant, 0 to 4095
#imm16 represents a 16-bit constant, 0 to 65535
operand2 represents the flexible second operand
{cond} represents an optional logical condition
{type} encloses an optional data type
{S} is an optional specification that this instruction sets the condition code bits
Rm {, shift} specifies an optional shift on **Rm**

* Arithmetic Instructions:

Flexible operand for $R_d = R_n + op2$

Op2 = ...

- Rm → `Add Rd, Rn, Rm`
- #Constant → `Add Rd, Rn, #constant`
- Rm <<n → `Add Rd, Rn, Rm, LSL#n`

if R_d is present R_d is destination, otherwise is R_n

→ Basic format: `OP{S}{cond} {Rd,} Rn, <op2>`

→ Immediate operand: `OP{S}{cond} {Rd,} Rn, #im12` (from 0 to 4095)

ADD, SUB, MUL

Suffix

if S is present, instruction will set condition codes (optional)

Condition

it used to optionally execution instruction & must be part of if-then block

EQ NE GT [ALL]

* Comparison instructions:

- they are used to create conditional execution such as (if-then, for loops, while loops)
- they don't save the result of the operation, but always set the condition code

① Compare Operations:

- `CMP{cond} Rn, <op2>` ;Rn - op2
- `CMN{cond} Rn, <op2>` ;Rn - (-op2)

→ they don't save the result but set the condition codes

② Test Operations (`TST{cond} Rn, <op2>` & `TEQ{cond} Rn, <op2>`):

→ these instructions test the value in a register against operand 2. They update the condition flags on the result, but don't place the result in any register

i. TST instruction performs a bitwise AND operation on the value in R_n & the value in operand 2. This is the same as ANDS instruction, except that the result is discarded

i. TEQ instruction performs a bitwise XOR operation on the value in R_n & the value in operand 2. This is the same as EORS instruction, except that the result is discarded

* Logical Instructions:

→ Basic format: `OP{S}{cond} {Rd,} Rn, <op2>`

- AND ;Rd=Rn&op2
- BIC (Bit clear) ;Rd=Rn&(~op2)
- ORR ;Rd=Rn|op2
- ORN ;Rd=Rn|(~op2)
- EOR exclusive or ;Rd=Rn^op2

* to Set a bit: Use ORR operation with a mask { x = x | (1 << i); }

(Position of the bit to be set)

①
```
MOV R0, #1
ORR R1, R0, LSL #i
```

②
```
ORR R1, #m
;m is the mask
```

* to clear a bit: Use AND operation with a mask { x = x &~ (1 << i); }

* to toggle a bit: Use EOR operation with a mask { x = x ^ (1 << i); }

* Shift Operations:

① Shift left (LSL ;Rd=Rm<<n): Logical Shift Left LSL, 0≤n≤31

→ the logical shift left works for both unsigned & signed multiply by 2^n

→ A zero is shifted into the least significant position, & the Carry bit will contain the last bit that was shifted out

② Shift Right (LSR ;Rd=Rm>>n & ASR ;Rd=Rm>>n):

→ the logical shift right is similar to an unsigned divide by 2^n

→ A zero is shifted into the most significant position, & the Carry flag will hold the last bit that was shifted out — Logical Shift Right LSR, 1≤n≤32

→ the Arithmetic shift right is similar to an signed divide by 2^n

→ Notice that the Sign bit is perserved, & the Carry flag will hold the last bit that was shifted out — Arithmetic Shift Right ASR, 1≤n≤32

③ Rotate (ROR ;Rd=Rm>>n & RRX ;Rd=Rm>>n):

→ the two rotate operations can be used to create multiple-word shift functions

→ there is no rotate left instruction — Rotate Shift Right ROR, 1≤n≤32; Rotate Right Extended RRX, n=1

* Memory Access Instructions:

empty 32-bit word
B unsigned byte
SB Signed byte
H unsigned halfword
SH Signed halfword

→ Basic format: OP{type}{cond} Rd, [Rn]

① load instructions:

→ Used to load data from memory into a register

→ [LDR{type}{cond} Rd, [Rn]]: load 32-bit number at Rn to Rd

→ [LDR{type}{cond} Rd, =value/label]: Same as pointer in C language

it's a special form of LDR to load a constant or address into a register

→ [ADR{cond} Rd, label ;set Rd equal to the address at label]:

it uses PC-relative addressing & is a handy way to generate a pointer to a constant in code space or an address within the program

② Store instructions: STR{type}{cond} Rt, [Rn] "Store 32-bit number at Rt to Rn"

→ Used to store data from register into a memory

③ Move instructions:

→ they get their data from the machine instruction or from within the processor & don't require additional memory access instructions

→ MOV{S}{cond} Rd, <op2> ;set Rd equal to op2

→ MOV{cond} Rd, #im16 ;set Rd equal to 16-bit immediate (from 0 to 65535)

→ MVN{S}{cond} Rd, <op2> ; set Rd equal to -op2

→ for small values (less than or equal 16-bit) we use MOV, but for greater values we use LDR

③ Stack instructions:

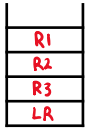
→ the stack push & pop instructions can operate on one register or on a list of registers

→ PUSH {Rt} ;save Rt on the stack

→ POP {Rd} ;remove from the stack into Rd

→ PUSH {R1-R3,LR} ;save R3,R2,R1 Respectively and link register (first item pushed)

→ POP {R1-R3,PC} ;restore to R1,R2,R3 and PC (top of stack is popped to R1)



* Branch Instructions:

① Unconditional branch:

→ B{cond} label ;branch to label

→ BX{cond} Rm ;branch indirect to location specified by Rm

→ BL{cond} label ;branch to subroutine at label

→ BLX{cond} Rm ; branch to subroutine indirect specified by Rm

② Conditional branch:

→ BEQ label ;branch if Z==1 (Equal)

→ BNE label ;branch if Z==0 (Not-Equal)

→ there's much of them

"ARM Assembly Instructions"

* Reset:

→ After reset

- the 32-bit value stored at location 0 of Flash ROM into the SP
- the 32-bit value stored at location 4 of Flash ROM into PC
- LR register value is set to 0xFFFFFFFF

* The Stack:

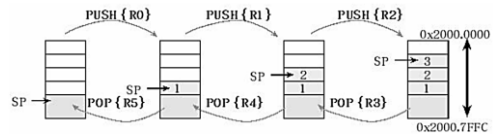
→ the stack is a last-in-first out (LIFO) temporary storage

→ we refer to the most recent item as the top of the stack, it's actually the item stored at the lowest address

→ the stack pointer (SP) points to the data on the top of the stack

→ SP points to the last item pushed, which will also be the next item to be popped

PUSH {R0}
PUSH {R1}
PUSH {R2}
POP {R3}
POP {R4}
POP {R5}



→ to push data on the stack, the stack pointer is first decremented by 4 & then the information is stored at the address specified by SP

→ to pop data from the stack, the information pointed by SP is first retrieved, and then the stack pointer is incremented by 4

* The rules one has to follow when using the Stack:

1. Functions should have an equal no. of pushes & pops
2. Stack accesses push or pop should not be performed outside the allocated area
3. Stack reads & writes should not be performed within the free area
4. Stack push should first decrement SP, then store the data
5. Stack pop should first read the data, & then increment SP

*Addressing Modes :

→ It's the format the instruction uses to specify the memory location to read or write data, a single instruction could exercise multiple addressing modes

- Register to register Addressing (Register direct) → `MOV R0, R1`
- Direct Addressing (Absolute) → `LDR R0, MEM`
- Immediate Addressing (literal) :

① `MOV R3, #30` ② `LDR R0, =const` (16-bit) ③ `ADD R1, R2, #0xFF` ④ `CMP R0, #22`

→ "`LDR Rn, =const`" it's a pseudoinstruction, the assembler fetches it then executes it as MOV if the constant is less than or equal 16-bit, otherwise the assembler uses PC relative addressing mode, So the constant decide which addressing mode is used

- Register Indirect with register offset (Double Reg indirect) :

→
```
LDR R0, [R1,R2]    ; R0 = Mem[R1 + R2]
                   ; R1 is unmodified
```

- Register Indirect with Shifted Register Offset (with Scaling) :

→
```
LDR R0, [R1,R2,LSL #2]    ; R0 = Mem[R1 + 4*R2]
                          ; R1 is unmodified
```

- Regular Register Indirect Addressing (Indexed, Base) :

→ it means that the location of an operand is held in a register, it's also called indexed addressing or base addressing → `LDR R0, [R1]`

- Register Indirect with Immediate Offset (Pre-indexed, base with displacement) :

→ it's a memory-addressing mode where the effective address of an operand is computed by adding the content of a register & an immediate offset coded into load/store instruction →
```
LDR R0, [R1,#4]    ; R0 = Mem[R1 + 4]
                   ; R1 is unmodified
```

- Register Indirect pre-incrementing (Pre-indexed, autoindexed) :

→ A pointer register used to hold the base address. An offset can be added to achieve the effective address →
```
LDR R0, [R1,#4]!
; R1 += 4
; R0 = Mem[R1]
; R1 is modified before loading
```
A.k.A pre indexed immediate offset

- Register Indirect post-incrementing (Post-indexed, autoindexed) :

Similar to pre indexed but it first accesses the operand at the location pointed by the base register, then increments the base address →
```
LDR R0, [R1], #4
; R0 = Mem[R1]
; R1 += 4
; R1 is modified after loading
```
→ A.k.A post indexed immediate offset

- PC Relative Addressing :

→ It's an addressing mode where the effective address is calculated by its position relative to the current value of the program counter

→ It's indexed addressing mode using the PC as the pointer

- `LDR R0, =imm32` → to move any 32-bit value into a register
- `LDR R0, [PC,#28]` → using PC register instead of general-purpose register
- branch to subroutine at label → it's used for branching & for calling functions

  `BL{cond} label`
- Jump to location using PC-relative addressing

  `B label`

* Test yourself → QUIZ