Subject: 1

89/9/16

ARM :

32 bit architecture

Byte = 8 bits
Halfword = 16 bits (2 bytes)
Word = 32 bits (4 bytes)

Most ARMs implement two instruction sets
- 32 bit ARM instruction set — خوب نیست چون خیلی وقت دستورها کوتاه‌ترند
- 16 bit Thumb instruction set

Jazelle cores can also execute Java bytecodes.
سخت‌افزار ARM — هر کدام از دستورها ۸ بیت است در Jazelle.

CPU می‌تواند switch کند بین مدهای ARM، Thumb، Jazelle. (ابتدا با 32 بیت می‌آید بالا).

Processor Modes:

معمولاً چند application هم‌زمان دارند انجام می‌گیرند

ARM has seven basic operating modes:

User: unprivileged mode under which most tasks run.
معمولاً CPU در این مد است (۸۰٪ موارد).

FIQ:
Fast Interrupt Request — سریع به وقفه جواب می‌دهد

IRQ:
Interrupt Request — تأخیر جواب به وقفه خیلی مهم نیست

Supervisor

Abort: CPU دست‌برد می‌زند به قسمتی از حافظه که مجاز نیست (MMU خبر می‌دهد از آن)

Undef: CPU به دستوری می‌رسد که جزء Instruction Set آن نیست.

System: privileged mode using the same registers as User mode.

IDEA

ARM Registers :

- 16 registers , 32 bits each
- 13 registers are general purpose (بدون ترس هر استفاده دلخواهی از آن می‌کنیم)
- R13 : Stack pointer
- R14 : LR : Subroutine Linker Register : stores return address
- R15 : Program Counter

  ARM : دو بیت کم ارزش همیشه صفر است

  Thumb : یک بیت کم ارزش همیشه صفر است

  Jazelle : بیت کم ارزش می‌تواند هر چیزی باشد
- CPSR : Current Program Status Register

  Contains condition code flags
- SPSR : Saved program status Register

  A copy of CPSR

CPSR :

Condition code flags

N نتیجه محاسبه‌ی ریاضی منفی

Z نتیجه محاسبه‌ی ریاضی صفر

C رخ دهد carry

V رخ دهد overflow

J : Jazelle State

T : Thumb State

Mode bits : specify the process mode

ARM Register Set :

مدهای مختلف برای خودشان shadow register دارند برای اینکه محتوای رجیسترهای اصلی تغییر نکند در آن مد. برای رجیسترهایی که shadow ندارند باید ذخیره در حافظه صورت بگیرد (back up).
هر مدی برای خودش SPSR دارد.

Note : system mode uses the user mode registers

Thumb State : فقط می‌توان از ۸ رجیستر استفاده کرد

Exception Handling:

When exceptions occur the ARM:

- Copies CPSR into SPSR_<mode>
- Changes CPSR
- Stories the return address in LR_<mode>

Byte Ordering:

ARM operates in two modes : Big Endian (بایت با ارزش‌تر در محل کم ارزش‌تر)
Little Endian

Instruction Sets :

MOV <cc> <S> Rd, <operands>

b2

flag را تغییر بدهیم یا نه

```
MOV CS  R0, R1     @ if carry is set then R0 := R1

MOV S   R0, #0     @ C, V unaffected
                   @ Z=1 , N=0
```

مقدار صفر را در R0 می‌ریزد.

```
ADD  R0, R1, R2    @ R0 = R1 + R2
ADC  R0, R1, R2    @ R0 = R1 + R2 + C
SUB  R0, R1, R2    @ R0 = R1 - R2
SBC  R0, R1, R2    @ R0 = R1 - R2 + C - 1
( RSB  R0, R1, R2  @ R0 = R2 - 1 )
```

معکوس

Bitwise Logic:

```
AND  R0, R1, R2    @ R0 = R1 and R2
OR                 @        or
EOR                @        xor
BIC                @ R0 = R1 and (~R2)
```

بیت‌هایی از R2 که یک است را در R1 صفر می‌کند.

IDEA

Subject: 4

89 | 9 | 21

```
CMP  R1, R2          @ set cc on R1-R2   : compare
  (compare)
CMN                  @  "   "   - R1+R2
  (compare negated)
TST                  @  "   "   - R1 and R2
  (bit test)
TEQ                  @  "   "   - R1 xor R2
```

flag ها را تغییر می‌دهد ولی مقادیر رجیسترها ثابت می‌مانند.

* Logical Shift left:

```
MOV  R0, R2, LSL #2   @ R0 := R2<<2
                      @ R2 unchanged
```

* Logical Shift Right

```
MOV  R0, R2, LSR      @ R0 := R2>>2
```

* Arithmetic Shift Right

اگر عدد مثبت باشد این شیفت با شیفت منطقی یکی است ولی برای اعداد منفی این طور نیست.

```
MOV  R0, R2, ASR #2   @ R0 := R>>2
```

* Rotate Right

```
MOV  R0, R2, ROR #2
```

شیفت می‌تواند همزمان با اعمال ریاضی انجام شود.

64 Bit Addition

```
ADDS  R2, R2, R0
ADC   R3, R3, R1
```

کوئیز:

مقدار ثابت R0 را در 37 ضرب می‌کنیم:

37 R0 = 32 R0 + 4 R0 + R0

```
MOV  R1, R0, LSL #2
MOV  R2, R0, LSL #5
ADD  R3, R1, R2
ADD  R0, R0, R3
```

Multiplication :

```
MUL  R0, R1, R2        @ R0 = R1 x R2
```

- R2 can't be immediate
- R0 and R1 can't be the same

```
MLA  R4, R3, R2, R1    @ R4 = R3 x R2 + R1
                       @ Multiply and accumulate
```

Multiply with a constant can often be more efficiently implemented using shifts

```
MOV  R1, #35
MUL  R2, R0, R1
```

or

```
ADD  R0, R0, R0, LSL #2    @ R0' = 5 x R0
RSB  R2, R0, R0, LSL #3    @ R2 = 7 x R0' = 8R0' - R0'
                           @ Reverse Subtract
```

MUL و MLA زمان بیشتری طول می‌کشد تا اجرا شود. (به دلیل ساختار سخت‌افزاری)

Data Transfer Instructions :

Moving data between registers and memory.

Three basic forms

- Single register load/store
- Multiple register load/store
- SWP (swap)

```
LDR   R0, [R1]      @ R0 := mem32[R1]
STR   R0, [R1]      @ mem32[R1] := R0
table:  .word 10
ADR   R0, table
```

Addressing Modes:

Three ways to specify offset:

~ Constant

```
LDR  R0, [R1, #4]      @ R0 := mem[R1+4]
```

خود R1 تغییر نمی‌کند اگر بخواهیم تغییر کند باید ! داشته باشد آخرش

~ Register

```
LDR  R0, [R1, R2]      @ R0 := mem[R1+R2]
```

~ Scaled

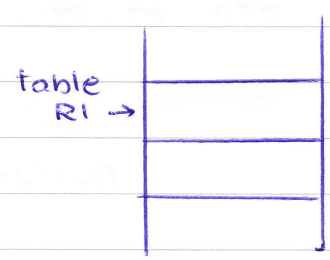
```
LDR R0, [R1, R2, LSL #2]
```

```
LDR  R0, [R1, #4]!     @ R1 = R1+4 , R0 := mem[R1]   (R1+4)
                       @ No extra time, fast
```

Pre-indexed addressing / Post-indexed addressing

بعد از اینکه access انجام شد می‌آییم R1 را تغییر می‌دهیم

```
loop:   ADR  R1, table
```



LDM → load multiple registers / STM → Store multiple registers

```
LDMIA  R1, {R0, R2, R5}
```

Subject:

suffix :

IA	increase after
IB	increase before
DA	decrease after
DB	decrease before

LDMIA $R_n, \{R_1, R_2, R_3\}$

LDMDA $R_n, \{R_1, R_2, R_3\}$

LDMIA R0, {R1 - R3}

R1 : 10
R2 : 20
R3 : 30
R0 : 0x10

add	data
0x00	10
0x014	20
0x018	30
0x01C	40
0x020	50
0x024	60

LDMIA R0!, {R1 - R3}

R1 : 10
R2 : 20
R3 : 30
R0 : 0x01C

LDMIB R0!, {R1 - R3}

R1 : 20
R2 : 30
R3 : 40
R0 : 0x01C

LDMDA R0!, {R1 - R3}

R1 : 40
R2 : 50
R3 : 60
R0 : 0x018

LDMDB R0!, $\{R_1 - R_3\}$

R1 : 30
R2 : 40
R3 : 50
R0 : 0x018

Subject:

Application : Copy a block of memory (32 bytes/aligned)

```
loop :   LDMIA   R9! , {R0-R7}
         STMIA   R10! , {R0-R7}
         CMP     R9, R11
         BNE     loop            @ branch not equal
```

Branch Conditions :

	B BAL	
*	BEQ	Equal
*	BNE	Not equal
	BPL	Plus (result was positive or zero)
	BMI	Minus
	BCC	Carry clear
*	BGT	Greater than
	BGE	Greater or equal
*	BLE	Less or equal

Branch and Link :

BL instruction saves the return address to R14 (LR) → Linker Register

```
BL   sub
  ⋮
sub: - - -
  ⋮
MOV  PC, LR
```

Stack :

Mode	LDM (POP)		STM (PUSH)	
Full Ascending (FA)	LDMDA	LDMFA	STMIB	STMFA
Full Descending (FD)	LDMIA	LDMFD	STMDB	STMFD
Empty Ascending (EA)	LDMDB	LDMEA	STMIA	STMEA
Empty Descending (ED)	LDMIB	LDMED	STMDA	STMED

IDEA

Subject:

89/9/23

ARM9 : DRAM Controller ⇒ DRAM را به میکرو وصل می‌کنیم
حافظه 64 Mbyte

flash حافظه‌ی : درازاست

Slot دارد برای اضافه کردن flash

USB → host
↘ peripheral

گوش کن :
دیتا در برنامه‌ی Assembly بنویسید که 64 داده 4 بایتی را از آدرسی که R0 مشخص کرده به آدرسی که R1 مشخص کرده، انتقال دهد.

* ARM Real View Development Suit. (RVDS)
Assembler/Compiler . مبتنی بر eclipse است.

* Keil MDK

* IAR Embedded Workbench

* Green Hills

* GCC : GNU C Compiler : Open Source رایگان.
از رنج خیلی بزرگ CPU ها پشتیبانی می‌کند از جمله x 86 , Power PC , ARM , AVR , PIC
انجام می‌دهد Cross compilation
برنامه‌ی ARM را کامپایل می‌کند چون روی platform x86 است پس Cross compile کرده‌ایم.

* Windows CE از شرکت Microsoft

Watchdog Timer : به صورت سخت‌افزاری می‌شمارد و وقتی به Threshold رسید reset می‌کند (برای مواردی که سیستم hang می‌کند) می‌توانیم آن را disable کنیم. ما هر چند وقت یک بار آن را صفر می‌کنیم تا reset نشود ولی اگر سیستم hang کرده باشد میکرو دیگر صفر نمی‌شود بنابراین پس از مدتی reset می‌شود میکرو.

IDEA

وقتی میکرو بالا فراخوانی باید initialization انجام شود که کافی است برای این کار که مربوط را در برنامه قرار دهیم.

برنامه نویسی به اسمبلی

```
__asm MyFunc ( unsigned int * scrAddr, unsigned int * dstAddr)
{ STMDB   R13!, {R0-R10}
label
  LDMIA   R0!, {R4-R7}        → MOV R9, #16
  STMIA   R1!, {R4-R7}
  SUBS    R9, #1 ⇝ R9,R9,#1
  BNE     MyFunc_loop
  LDMIA   R13!, {R0-R10}
  MOV     PC, LR ⇝ BX <rn>
}
```

label = MyFunc_loop

flag ها set می‌شوند

```
int main (void)
{ int i;
  unsigned int scrArray[64], dstArray [64];
  for(i=0, i<64, i++) scrArray[i] = i*3;
  MyFunc(scrArray, dstArray)
}
```

Branch & Link

```
BL  sub1          @ call sub1

sub1 :  STMFD R13!, {R0-R2, R14}
        BL sub2
        LDMFD R13!, {R0-R2, PC}
sub2 :  ...
        ...
        MOV PC, LR
```

زیرا دو sub در دل هم وجود دارند

Conditional Execution:

```
CMP R0, #5
BEQ bypass        @ if(R0! = 5)
ADD R1, R1, R0    @ R1 = R1+R0-R2
SUB R1, R1, R2
bypass:
```

Subject:

این برنامه معادل برنامه‌ی قبلی است که branch دارد ولی این برنامه کوچکتر و سریع‌تر است

```
CMP     R0, #5
ADDNE   R1, R1, R0
SUBNE   R1, R1, R2
```

Rule: if the conditional sequence is three instructions or less we use conditional execution.

مثال دیگر:

```
if ((R0 == R1) && (R2 == R3)) R4 ++
    ...
    ...
    ...
    ...
```

Instruction set:

BIC : Bit Clear

RSB : Reverse Subtract (سوم منهای دوم)

SUB : Subtract (دوم منهای سوم)

TEQ : Test Equivalence (از حاصل XOR flag ها را تنظیم می‌کند)

TST : Test (از حاصل AND flag ها را set می‌کند)

CMP : Compare (می‌توان مقدارها را با CMP انجام دهیم)

ARM assembly program

label	Operation	operand	comments
main:	LDR	...	@
	STR	...	
الزاماً نیست	SWI	#11	

ADR R0, value1 ⇝ ADD R0, PC, #40

```
value1:  .word  0x00000001, 0xF0000000
value2:  .word  0x00000000, 0x10000000
result:  .word  0
```

Loops:

for(i=0; i<10 ; i++) {a[i]=0;}

```
        MOV   R1, #0
        ADR   R2, a      → آدرس a را در R2 ریخته
        MOV   R0, #0     → i مکان
LOOP:   CMP   R0, #10
        BGE   EXIT
        STR   R1, [R2, R0, LSL #2]
        ADD   R0, R0, #1
        B     LOOP
EXIT:   ...
```

while loops:

```
LOOP: ...
      BEQ   EXIT
      B
EXIT
```

Find Larger of Two Numbers:

```
      LDR   R1, value1
      LDR   R2, value2
      CMP   R1, R2
      BHI   Done
      MOV   R1, R2
Done:
      ...
```

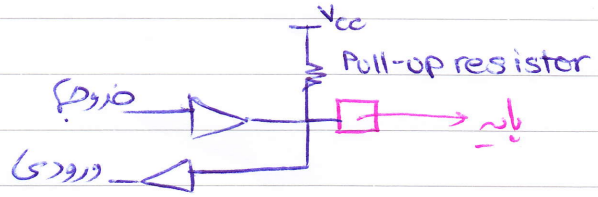
i--=j به اندازه j از i کم شود

ما برنامه را همیشه به C می‌نویسیم و فقط گاهی اوقات لازم می‌شود که از assembly استفاده کنیم.
(مثلاً وقتی یک حلقه قرار است خیلی تکرار شود و می‌خواهیم Optimum نوشته شود یا وقتی که به یک رجیستر خاصی می‌خواهیم دسترسی پیدا کنیم.

Parallel Input/Output Controller :

peripheral که اجازه می‌دهد هر مقدار داده‌ای را روی پایه‌ها قرار دهیم.

سه حالته
توسط رجیسترها طرز کار آن مشخص می‌شود



Pull-up : وقتی پایه درایو نشده پایه حتماً یک (یا Pull-down، صفر) تولید می‌کند

Glitch Input Filter Enable Register : Glitch تغییرات سریع سیگنال که معمولاً نامطلوب هستند فیلتر شوند.