* كل ما كانت cache أقرب لل CPU بتكون أسرع.
* ال cache وال main memory بنسميهم Volatile وهدول بفقدوا المعلومات عند قطع الطاقة، مش زي NOM-volatile (فلاش/ هارد ديسك...).

Networks:
local area network (LAN): Ethernet
Wide area network (WAN): Internet wireless network/wiFi/Blue tooth

---

* في الجيل الأول من الحاسوب كانت يستخدم VaccumTubes بدل الترانزستور، وكانت ضخمة جدًا تنتج حرارة كبيرة جدًا تنتج تعتيمة من 1945 لـ 1952 على يد John von Neuman
* Fundamental design approach was the Stored program concept.

* في الجيل الثاني تم استخدام الترانزستور، وكانت أرخص وأصغر وأقل استهلاك للطاقة من Vacum. 1947. وأصبح يقوم بعمليات أكثر تعقيدًا ويستخدم High-level-lang، وأصبح يقوم بتمثيل برامج ونقل معلومات.

* في الجيل الثالث تم استخدام (IC) Integrated Circuit وهو يتكون من عدة ترانزستورات، مما ترك، طبعًا أول ما صنعوه كانت عدد الترانزستور فيه قليل، بعدين مع الوقت طوروه أنهم يزيدوا العدد بنفس المساحة.

Moore's law: هو أن عدد الترانزستورات على الرقاقة الواحدة يتضاعف كل سنة.

الـ CPU الحقيقي اللي فيه core واحد بس يعني CPU عادي، أما MultyCore CPU يعني أكثر من Microprocessors أو processors.

* يوجد نوعان من أنواع الـ processors: ~~ARM~~ RISC و ~~intel x86~~ CISK

intel x86 ← بيستخدم Complex instraction set computer (CISC)
يعني حاولوا يخلوا التعقيد في الهارد وير يعني. خلوا كل العمليات المعقدة وليس فقط البسيطة تتعمل فيه يعني بتكون حجم العمليات فيه كبير (اللي بتقوم) وبالتالي برمجتها hard.

ARM ← بيستخدم Reduced instruction set computer (RISC)
يعني عدد العمليات اللي بتقوم فيها قليل وتقتصر على العمليات البسيطة يعني حاولوا يبسطوا الهاردوير أقصى ما يمكنه و يعتمدوا على السوفتوير.

microcontroller: هي قطعة يكونه موجود فيها memory/IO/CPU ولكنه بأحجام بسيطة (تقوم بالتحكم بالأجهزة).

Embedded Systems: هي الأجهزة اللي تتابع للتحكم وتستخدم microcontroller فيها، مثل الغسالة/مايكروويف/كاميرات.

The Internet of Things (IOT): هو أنو يعني تشبك الأجهزة على النت وتصير عتواصل مع الأجهزة عشان يعد، سواء تتحكم أو مشاهدة.

CH3:

* أي مشكلة بتواجهني بقدر أحلها بطريقتين:
أما عن طريق Software أو Hardware.

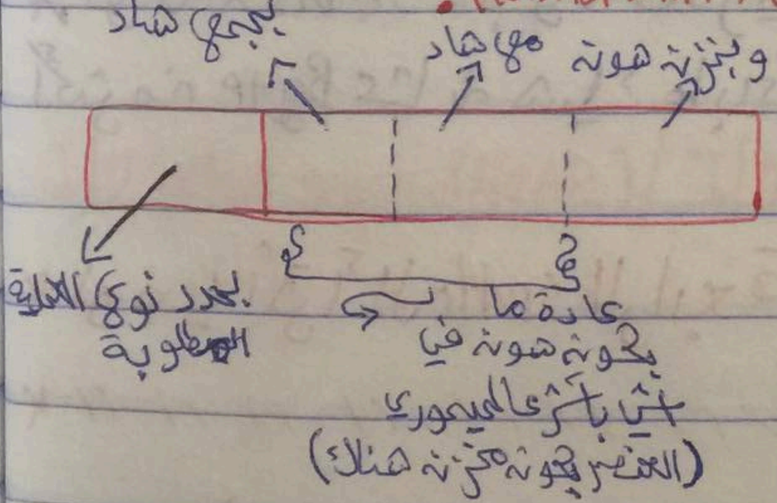
* الكمبيوتر بقدر اعمل لأي اشي بدي اياه، فمثلاً بقدر اعمل عليه Software وأقلبه بمير راوتر، وبقدر اجيب Hardware فيها Control unit بتشتغل راوتر.
* بس إنا طلب مني برنامج محاسبة، ما بزبط غير Software.

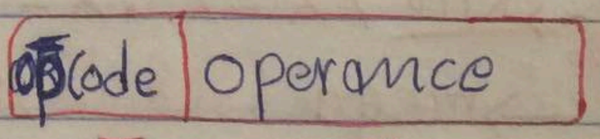
Figure 3.1

Software: A sequence of codes or instructions.

.Part of the hardware interprets each instruction and generates control signals.

* الـ CPU بتترجم الـ Instruction، والـ Hardware يكونه معهم أنواع هاد الـ Instruction بتحولها لـ Control Signal (control unit).

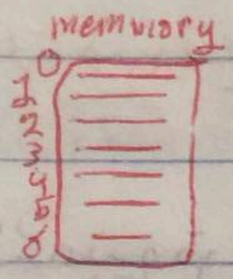




Instruction

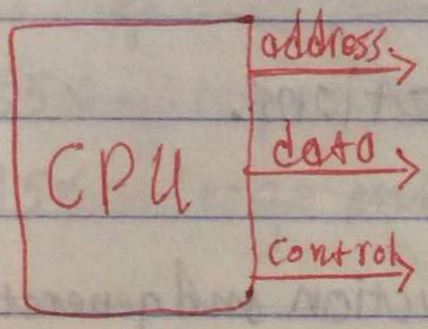
* طبعاً كل CPU بتختلف عن التانية الـ Instruction فيها.
* بكونه داخل الـ CPU ⟸ Instruction interpreter وهو مترجم الـ Instruction بدل الـ Assembler.

معظم الداتا اللي بنشتغل عليها بتكون بالميموري.



كيف الـ CPU بتقرأ من الـ Memmory؟

* الـ Memmory بتكون مقسمة لعدة cells حسب حجم الـ Memmory.
* الـ CPU مشبوكة مع memory عن طريق 3 buses : address bus, data bus, control bus



* طبعاً حجم الـ address هو اللي بحدد عدد الـ cells اللي بقدر بتعامل معها لو كانت 4bit بتكون قادر عالتعامل مع 16 cells مثلاً. $2^4 = 16$
* يعني لو الـ memory فيها أكتر من 16 عالفاضي.

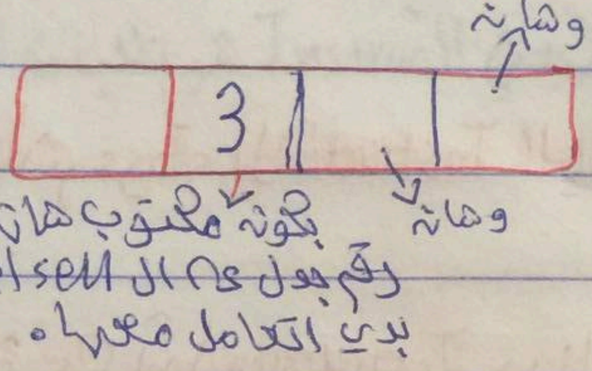
* لو كانت 32bit بتكون قادر بتعامل مع 4G cells ممكن تكون الـ cells 4G Bytes أكتر من كتابة هيك ما بتكفي. 4GB
$2^{32} / 2^{30} = 2^2 = 4G$
2^{10} → kilo
2^{20} → mega
2^{30} → Gega
2^{40} → Tera

* لو بدي أقرأ الـ cells الرابعة من 4bit address 0011 <= 4bit
00011 <= 5bit

هسا بعد معرفت الـ cell اللي بدي اتعامل معها من الـ address بنزل الـ Control Bus الـ لازم أعرف شو بدي أعمل معها هل بدي أقرأها و لا أكتب عليها؟
اللي بحدد

* عرفت انو بدي اقرأ من الـ Memory كيف رح انقل الداتا من الـ CPU؟
عن طريق الـ Data Bus.

* هيدا الـ CPU بعد يقرأ بخزنه مؤقتاً في الـ registers وبكل شغل عنا نشوف شو بدو يعمل.



* الـ CPU بكون فيها كتير rigesters بس مش كلها بتكون مشبوكة مع الـ Buses اللي بتكون مشبوكة مع الـ Buses بشكل بشكل بشكل Address

* Memory address rigster (MAR)

(MBR) Memory buffer register (MDR) Data

* اللي بتكون مشبوكة مع الـ Data Bus بشكل بشكل

لما بتعامل مع Input/output بكون الـ Inputs و كل Input/output بكون الو address مختلف وكل outputs بكون الها address مختلف و زي كأنها cells بالميموري.

فبنستخدم: I/O address register (I/O AR) / I/O buffer register (I/O BR)

* (I/O AR): تحدد I/O معينة.

* (I/O BR): تستخدم للتغير في الداتا بينه I/O و الـ CPU.

PC: Program Counter.
IR: Instruction register.

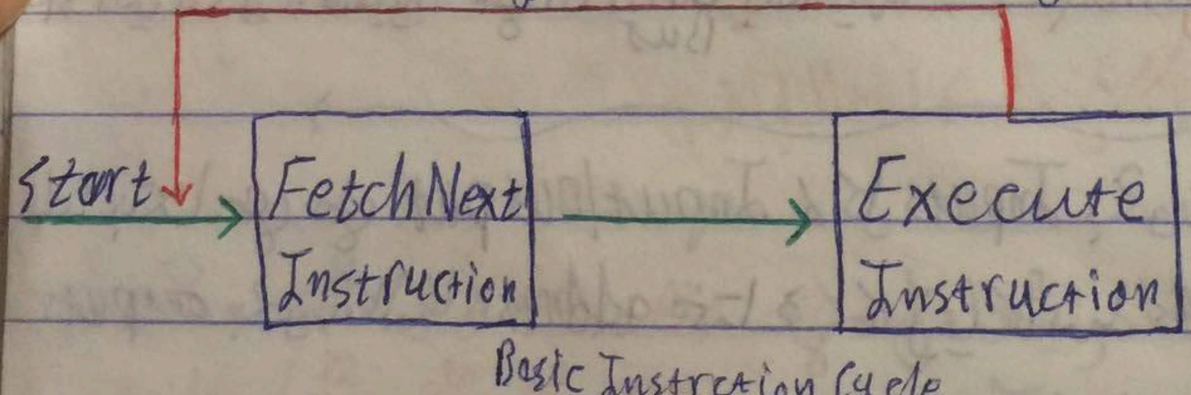
• PC ← register ⟵ CPU يوجد أيضاً في ال
IR ← register ⟵ CPU يوجد أيضاً في ال

* ال CPU بتبلش بتنفذ ال Instruction بالترتيب من الأول للآخر و ال PC بتعد وين وصل.

* بدايةً ال PC هي ال pointer اللي بتأشر عرقم ال Instruction اللي بدو يتنفذ.

* أول ما بتبلش بكونه ال PC بتأشر عـ أول Instruction بدنا ننفذ و بعدين PC بزيد.

* لما ال PC تأشر عـ Instruction التالي بكون جاهز لما Auto Increment



Basic Instruction Cycle

* كل الخطوات اللي بتصير عشان تنفذ Instraction واحد بنسميها Inst. Cycle

* لو بدنا نختصر شاي ال Cycle بنختصرها بخطوتين رئيسيتين.

Fetch Inst. / Excute Inst.

* Fetch هو اللي بجيب وبقرأ من الميموري. بنفذ ال Instruction Execute.

* إذا الرسمة تلتبر loop تبقى تنفذ حتى آخر Instruction.

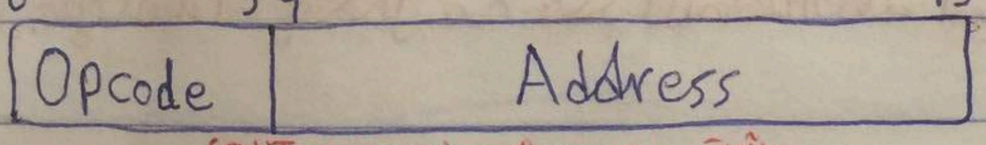
IR

* الـ fetch بعد متغير بتخزنه مؤقتاً في Instruction register و الـ processor بصير جوا هاد الـ Instruct داخل الـ register.

Action Categories: (يعني الـ Instruct شو بتطلب من الـ CPU)

* Processor-memory: بتنقل المعلومات من الـ procesor لل memory أو العكس أو من pro لـ pro أو من memory لـ memory.
* Processor-I/O: تنقل المعلومات من الـ Inpute لل procesor أو من الـ procesor لل out put.
* Control: بتحدد التسلسل في الـ excution لـ Instraction. زي Jump يعني loop/if.
* Data processing: تنفذ Arithmetic & logic operation على الـ Data.

مثال على آلة بسيطة:



(a) Instraction format للآلة

→ Accumulator 1 bit Address

* الـ opcode <= 4bit <= يعني بقدر يعمل 16 عملية حسابية أو أقل.
* من bit=>4 لـ bit=>15 هاي بتحدد الميموري أدور عليش ما في الـ operance وحدة؟
* هاي الآلة ما فيها الا register واحد (AC) و معناها ما بلزمني إلا (لـ) لـ كيف بتشتغل هاي الـ Instruction؟ لو ما كان موجود بتشغلو على AC دايماً السور الأول بكون موجود على الـ AC والسور الثاني بكون عنوان في الـ Address يعني لو بدي اجمع رقمين أو اي عملية أخرى و بخبي الرقم الأول في AC والرقم الثاني في Address و بخزن النتيجة على AC لأنو ما في register غيرها.

0001 = Load AC from Memory
0010 = Store AC to Memory
0101 = Add to AC from Memory

الصورة "10" أقصد.

الصورة 20 فيها وسعة Instruction cycle الخطوة بتفصل اليا بعير داخل fitch و exc.

كل Step عبارة عن State و كمان انتقل State ده Clock cycle بدي State.

عم فرض عندي ال CPU شغال بأي معينه و دخل عندي Input عال CPU، كيف ال CPU بدو يعرف انو اجا Input؟

في Interrupt Controller بكونه داخل ال CPU و بخلي ال CPU تكمل ال Instruction اليي بشتغل فيه وبعدينه تروح تشوف ال Input اليي دخل، بعدينه ترجع ال CPU بتكمل منه عند ال Instruction اليي وقفت عندو.

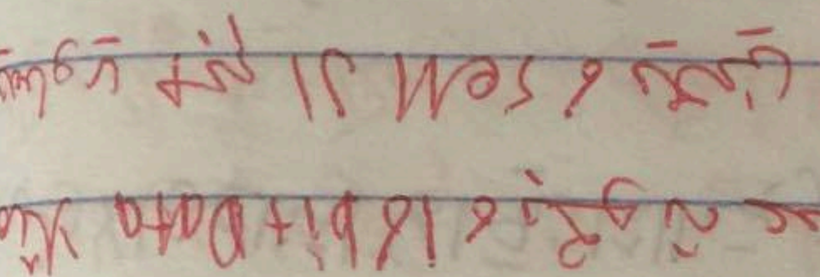
مش بس الهاردوير اليي بعمل انترابت ممكنه السوفتوير كمانه، لنفرض عندي برنامج شاغلوا إذا بدي ضغط بطلعلي اكسبشنه و بوقف البرنامج مثلاً (هاد انترابت)، و بطلعلي اكسبشنه و بكمل البرنامج (هاد برضو انترابت) و انا بحدد ال انترابت شو يعمل، بحدد كيف بحدد عن طريق اشي اسمو Interupt Service Routine (ISR).

بعمر تايمر Timer مع ال CPU و بخلي ال Timer كل وقت معينه يعمل انترابت لل CPU لأشي معينه.

عم فرض ضرب عندي اشي بالهاز، مباشرة بتعمل انترابت لل CPU، ممكنه أكتب السيرفس روتينه انو يطفي الهاز، أو ممكنه أخليه يطبع للمستخدم انو يطفي.

Computer modules:

* All words are selected.



* All words have a unique address.

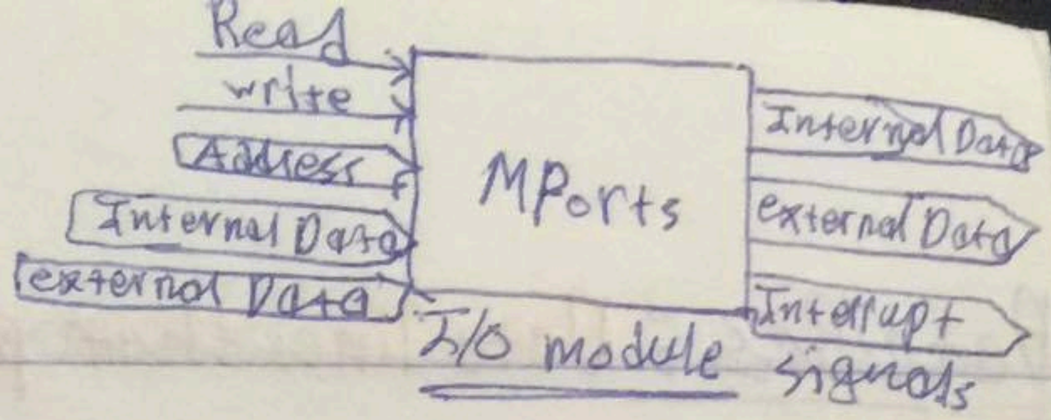
* Bus is a group of wires transfer data in form of signals (bits) between CPU and memory.

* Bus width is the number of bits can be transfer at once (Bus width is the same as the word size).

* Bus consists of data bus, address bus and control bus.

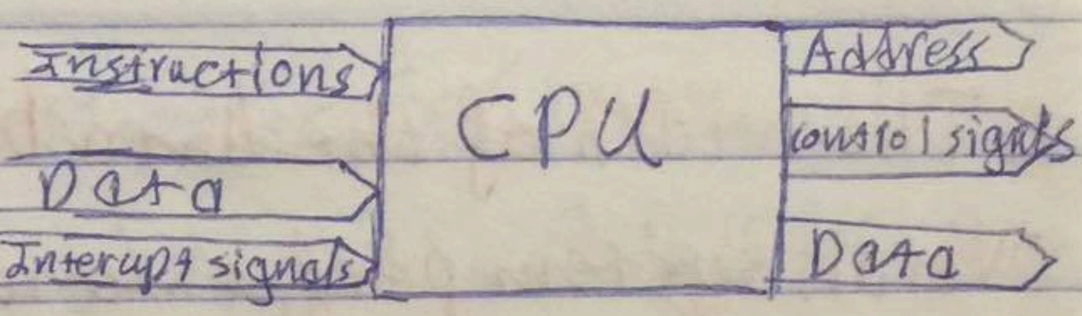
* Address bus is unidirectional, data bus is bidirectional.

* Control bus is a group of wires used to control access to the bus and data on the bus. CPU sends signals on the control bus to enable the peripherals. Peripherals can send signals to request direct memory access (DMA).



* كل الـ I/O بالكمبيوتر الها Address و مسافة كل Port الها Address و يعني كل I/O بتكونه الو Port Address.
* الـ Internal Data Input هي اللي بتكونه من الـ CPU و الـ external بتكونه من الجهاز ة (الكيبورد مثلاً). الباقي تعني الميموري.

* هُنا الـ Data Bus هو واحد بس بكونه مشبوك مع الـ CPU و الميموري.
* و الـ Address كذلك و وكلها.



The interconnection must be support the following types of transfers:
Memory to processor (load): Processor read an instruction or a unit of data from memory.
Processor to memory (store): Processor write a unit of data to memory.
I/O to processor: Processor reads data from an I/O device via an I/O module.
Processor to I/O: Processor sends data to the I/O device.
I/O to or from memory: An I/O module is allowed to exchange data directly with memory, without going through the processor using direct memory access.

* هون الـ Bus الها 3 أنواع.

DataBus: Data lines that provide a path for moving data among system modules.

* The number of lines determine how many bits can be transferred at a time.

* The width of the data bus is a key factor in determining overall system performance.

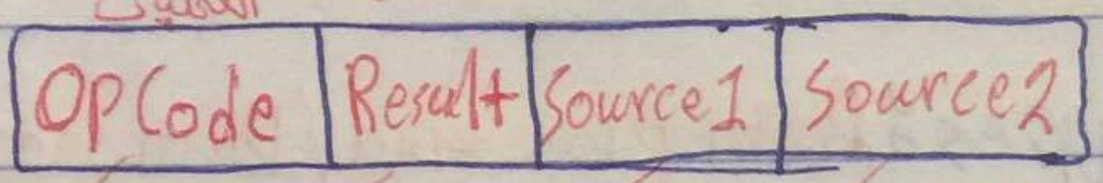
الـ Buses بتوع توزيع شبكة الـ (Address Bus, Control Bus) 80 و 90 سلك/سلك (بتوع توزيع شبكة الـ module).

مشكلة الـ Shared Buses انها بتعمل عندي تفاوت في السرعات ما بين الـ CPU وما بتشتغل عليه مع شئ واحد (الـ Bus).

Point-to-Point Interconnection:

حلوا بشبكة الأشياء السريعة مع بعض عشان الـ Bus ما يزيد الـ CPU
وبشبكة الأشياء البطيئة مع بعض زي الـ I/O
والـ cash وبارو بشبكة الأشياء البطيئة مع بعض زي الـ I/O.

ch12: Elements of a machine Instruction



Add R_0, R_1, R_2

* عشان فرضنا عندي 4 registers ومعناها بدي 2 binary bit حتى اقدر
00 R_0
01 R_1
10 R_2
11 R_3

* لو كانت عندي الماكينة بتعمل 64 oper معناها بلزمني 6 bit Opcode
* الـ 5 bit ما بيخلوا الا تنين بعطوني لحد 32 oper.

* الـ operance (Data) اللي بدي اشتغل عليها بتكون يا بـ register او بالميموري او I/O او Constant. Add R0, R1, 100

* لو كانت عندي الـ 8 bit Source2 و بدي أخزنه جواه const و شو أكبر رقم بقدر أخزنوا فيه؟
unsigned number $(0-255) <= 2^8 - 1$ عشان بنلبش منه سالب

signed num $(-128 - 127) <= -2^7 \rightarrow 2^7 - 1$

بالأسمبلي بنستخدم الكلمات والاسمبلر بحولها للغة الآلة.

ADD: Add
SUB: Subtract
MUL: Multiply
Div: Divide
LOAD: Load data from memory
STOR: Store data to memory

4	4	4	8
0001	0011	0001	00010110

مثال للتوضيح Add R3, R1, 16

Instruction Types: Data processing / Data storage / control / Data Movement

مشروح عنهم قبل ⟸ (الـ Inst شو بتطلب من CPU)

Design بطرق مختلفة Instr الـ ممكن نعملها

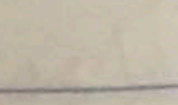
1) 2) 3)

مرورة 120

يوجد أيضاً Zero-Address وبكون بس operande وبعتمد عـ الـ stack

لما بدي اعمل عملية مثلاً ADD، بروح بضيف الـ (Top-1) عـ الـ Top وبخزنه النتيجة بال Top

push A
push B



Ex: $y = \frac{A-B}{C+D*E}$

push A	push D	push C	pop Y
push B	push E	ADD	
sub	MUL	DIV	

3-Address-Inst

ADD	100	R0	R1

⟹ بجمع R0 مع R1 وبخزنه النتيجة في cell (رقم 100)

2-Address-Inst

ADD	100	R0

⟹ بجمع cell رقم 100 مع R0 وبخزنه النتيجة في cell رقم 100

1-Address-Inst

ADD	100

⟹ بجمع cell رقم 100 مع AC (Accumulator register) وبخزنه النتيجة في AC

→ ADD [100], [200]

* في ال Intel ممنوع اعمل عملية على قيمتين ويكونوا مخزنين بالميموري

ADD [100], [200] : فبتعمل هيك ↓ لازم واحد أو مغير

mov R0, [200]

ADD [100], R0.

* في ال Arm ممنوع يكون واحد ميموري فبتعمل هيك :

LDR R1, [100]

LDR R2, [200]

ADD R0, R1, R2

STR R0, [100]

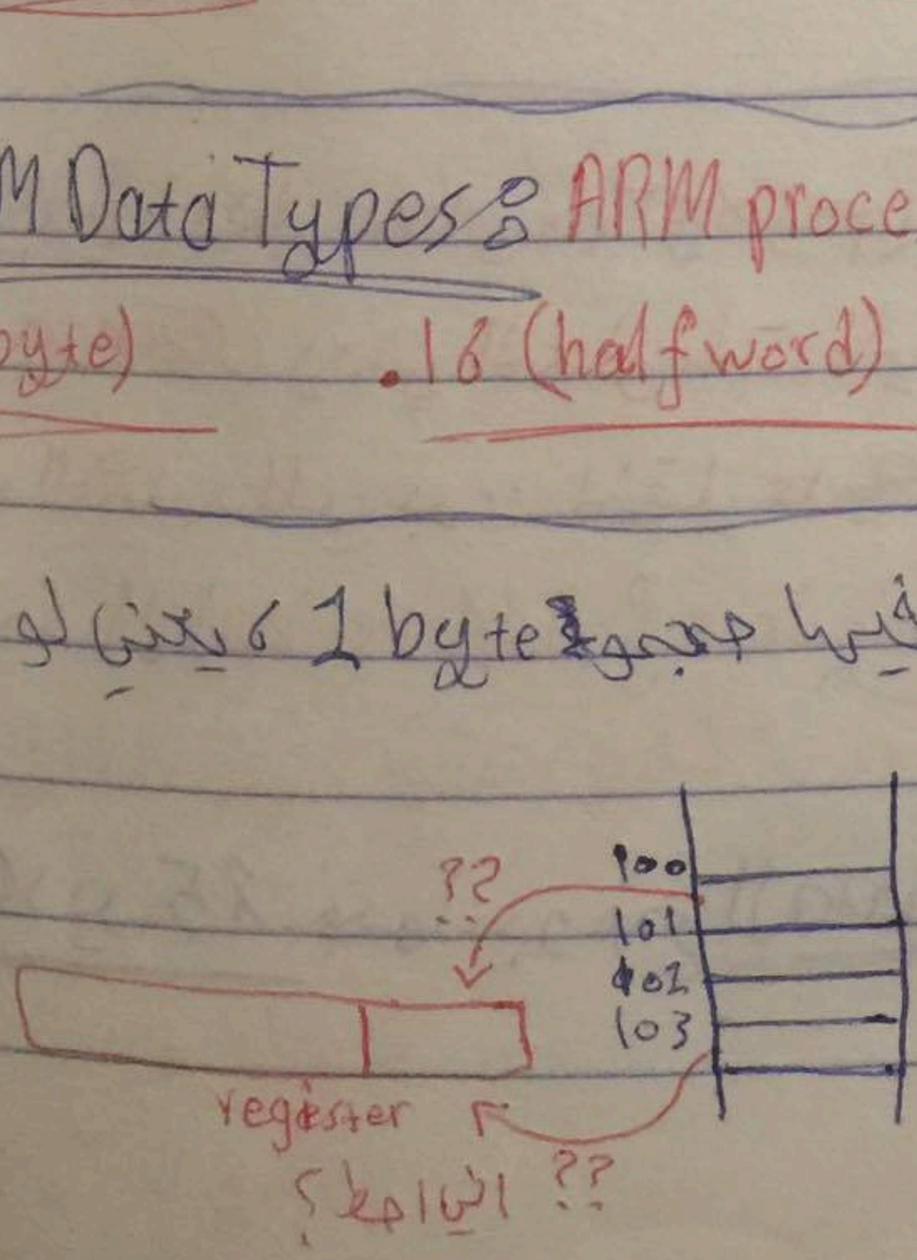
Types of operances: Address, Numbers, characters, logical Data

logical Data: 0 1 0 0 1 → False (0), True (1) بتتعامل معهن كل رقم لحالو ال

ARM Data Types: ARM processors support data types of:

- 8 (byte)
- 16 (halfword)
- 32 (word) bits in length

* الميموري بال ARM بتكون كل cell فيها حجمو 1 byte ، يعني لو بدي أخزن 32 bit num رح اخدلو 4 byted



جميعاً كل الريجسترات بال ARM 32bit

~~32bit~~

لابدي امزنة الرقم داخل register وقع تمير عندي مشكلات Endian لما
لأنو ما رح اعرف مين اخذ بـ least significant bit الـ byte الأولى و MSB آخر قيمة.
* طبعاً الثنتين مع بعض Design مختلف آخر بتختلف.

يعني الأولى (100) في الشمال اي قبل

* ولما تكون الـ least هي الـ byte إلي عند الـ lowest Address بنكي عنها Big Endian و لما العكس بتكون little Endian.

* الـ Intel دايماً بتتبع بس الـ little Endian و الـ ARM تدعم الثنتين.

* بال ARM لما تكون الـ status register E-bit تساوي صفر بيشتغل little و لما تساوي واحد بيشتغل Big. صورة 140 بتوضح الأخير

* في جدولين بسلايد 22/23 اطلع عليهم.

Data Transfer: * لابدي انقل معلومات لازم أول شي احدد الـ (source) يعني المعلومات اللي بدي اتعامل معها و الـ (destination) يعني العنوان اللي بدي انقل عليه و احدد كمان الـ (length of data) و احدد الـ (mode of Addressing). يعني طريقة كثيرة

* سلايد 25 جدول بوضح الـ length data

طبعاً بال shift القيمة اللي بتطلع بتكون carry flag.

shift and Rotate Operations: logical shift (unsigned number)
Arithmtic (signed number)

طبعاً بختلفوا الثنتين عنه بعض لأنو بال signed اول bit عاليسار بتحدد اذا رقم

بال logical لما بدي اعمل Right shift بضيف صفر منه اليسار وأول bit منه اليمين بتطلع و لما بدي اعمل left shift بضيف صفر منه اليمين واليسار بتطلع.

* بال Arithmetic لما بدي اعمل Right shift بضيف نفس قيمة الـ sign كانه مرة منه اليسار وأول bit منه اليمين بتطلع و لما بدي اعمل left shift بضيف صفر منه اليمين والقيمة اللي قبل الأخيرة ~~بتطلع~~ منه اليسار بتطلع لأنو الأخيرة sign.

Intel طريقة — ARM طريقة

* بتقدر انطلع القيمة الأخيرة عادي بدل القيمة اللي قبل الأخيرة وبتوقف إذا كانت الـ sign (آخر bit منه اليسار) 0 ومارت 1 أو العكس بكونه في overflow.

Rotate: * الـ Right Rotate بتنقل أول bit عاليمين عأول bit عاليسار
* الـ left Rotate العكس.

صورة 150 فيها امثلة.

طبعاً حكينا 8 bits عـ 32 آخر قيمة بتطلع حتى تكون الـ carry.

7 4 3 0 → 14
0111 → 1110

7 3,5 3
0111 → 0011

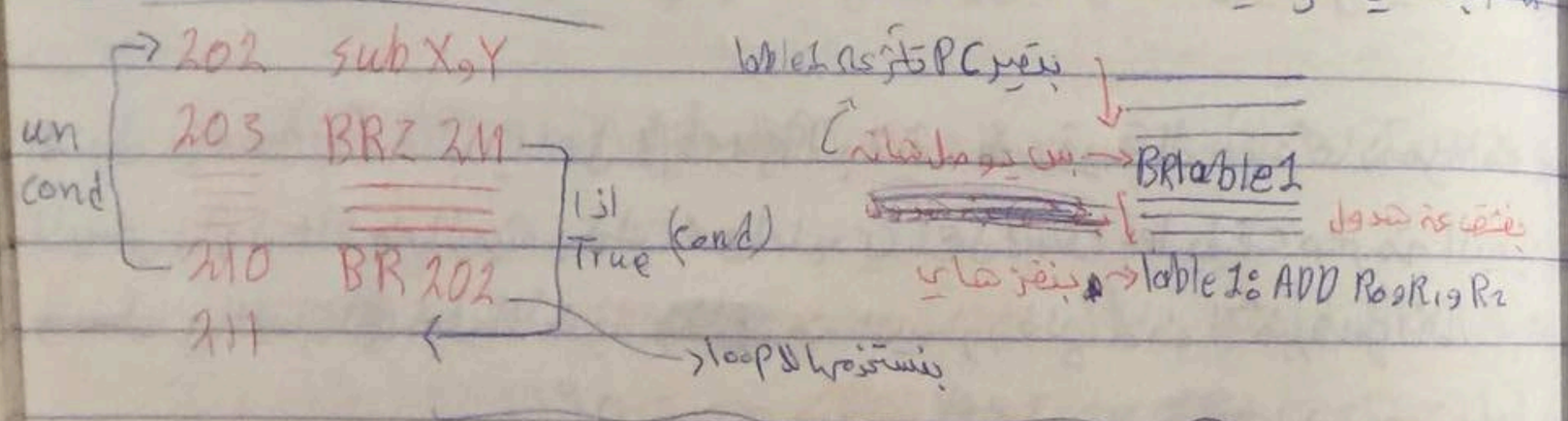
لما نعمل left shift كأنا بضرب بـ 2
لما نعمل right shift كأنا بقسم عـ 2

Conversions: Instructions that change the format or operate on the format of data.

Transfer of control: (Jumb/Branch) if else

1) unconditional Branch

* غالباً في نوعين من الـ Branch:



2) conditional Branch: Branch بتعمل (True لـ) بتحقق بتكون في شرط وإذا تحقق

flag الـ Os الـ condition متحققتش بتكونوا بيرو الـ

BRZ 211: 1 بتاوي Zero flag إذا الـ Branch يعني

211 على الكتري

BRE R1, R2, 235 → Address 235 Os لما R2=R1 يعني إذا

100 R1=-16

Skip Instruction:

Example is the increment-and-skip-if-zero (ISZ) instruction

ISZ R1 → بتزيد R1 واحد واذا ما بتاوي

BR 100 تاوي بتغير ما بتغير الى هنا

Procedure call Instructions: * زي الـ Service Routine

* بتاني بكون عندي اجزء من كود بستخدمو كتير ف قبل ما أكتبو بعمل call.

* في فرق بين الـ Branch و بين الفرق بينهما انو لما نعمل call وبتغير الكود بيروح بتكمل من محل ما عملنا call وبتكمل عادي متل زي الـ Branch.

* لما نعمل call عند 110 قيمة الـ instruction التالية (111) بتتخزن داخل stack و بعدين بتروح عال call وبتنفذ وبتكون في Return بتعمل pop stack PC الـ Os

مثال صورة 180

والـ stack كيف بيصير يوخذ الشيء بالترتيب عشان عندي أكثر من Procedure

صورة 180 *	صورة 170 * الـ Flag في Intel
* لما نستخدم A/B unsigned بس	* هنا "كل" موجودة بال ABM ما عدا AF/PF
* لما نستخدم O signed بس	- signed الـ over flow / * unsigned الـ carry number

* بالأسمبلي ما بيجوز اتعامل مع الميموري زي هيك ADD R0, [1101]
* بس بال Load (LDT) و بال Store (STR) بيجوز اتعامل مع ميموري

LDR R0, [110] ← بدلو عن الميموري
LDR R0, [R1] ← يعني الريجستر R1 بدا خلو ادرس لميموري

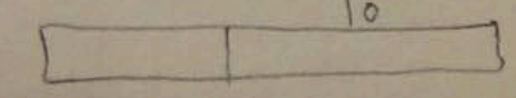
طب ليش مرات بكونه الأدرس مخزنه بـ register مش مكتوب بالأنستركشن
عن طول؟ لأنو ~~حجم~~ الـ length تاع الميموري آدرس بالأنستركشن
بكونه مش كتير كبير يعني عن فرضت لو كانه عندي الـ length=20 bit
و بكونه قادر يعمل Access لـ 2^{20} cells بالميموري [ADDRESS]
وكمان كل ما زاد حجم الأنستركشن بتصير ابطئ.

فإحنا بنروح بنخلي الأدرس بـ register لأنو الـ register حجمو 32 bit
فبالتالي بكونه قادر اتعامل مع cells أكتر و كمان بدا اتوفر عدد
الريجسترات مش كتير كبير فبالتالي بكونه الـ length كلو بالريجستر
قليل [IR] فبصير أسرع.

صورة 220 فوائد و امثلة أنواع الـ Addressing modes.

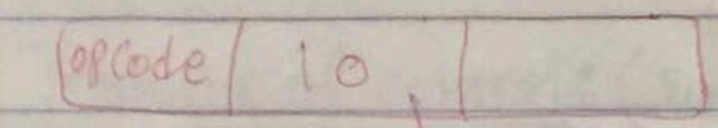
* فيزيا Intel بتدعم جميع الأنواع و الـ ARM مش كلهم.

* في الـ Immediate Addressing و الـ length هو اللي بحدد أكبر و أصغر رقم
يعدر اتعامل معو يعني الـ 10 bit ← (-512 - 511)



صورة 190 / صورة 200 .

CH13: Addressing modes



مثلا الـ 10 يعني register رقم 10 و الأدرس ميموري رقم 10 و الـ const كيف بدي آعرف
* الـ Address mode هو اللي بحدد.

أدرس ميموري
أو
* لو فرضت انو الـ 10 معناها register رقم 10 و طب هل الداتا موجودة
مباشرة بالـ register و لا فيها أدرس ميموري بدل عالداتا. Indirect
Direct

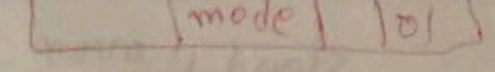
Addressing modes: Immediate / Direct / Indirect
Register / Register indirect / Displacement / stack

Immediate: ← لما يكونه رقم الـ const بتجيبوا من مكانه

صورة 210 بتوضح الأنواع.

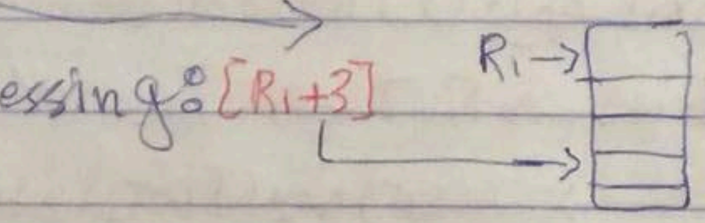
* لو كانه عندي هاد الـ Inst [101] وكنت بعرف انو register

وبدي INCR5 و كيف بدي اعرف انو اعلى Increment للـ register نفسو و الداتا اللي فيه و لا الميموري سيل و كيف بدي أميز
بنط الـ mode بتس



MOV R0, #100 ← # ← const بتدل على

Displacement Addressing: [R1+3]



[R1+SI] ← ممكن شرط register ثانية const

Relative Addressing: لو كانت عندي مجموعة اتنين تركيز

لو كانت معنى ال BR انو اروح على الاتنين تركيز

PC = 10 ← Absolute Address و 10 رقم بايونة

99
100
101
BR 10

Relative address لو كانت قيمة ازيد عن ال PC الحالي 10 بايونة

فبروح بنط عن 111 لأنو ال PC بصير اوي 111 (أول مبلش ننفذ

بصلح طول PC بصير اما بعدينه بنزيد ما بصير PC = 111 .

mode bit ال BR 10 ال

Intel ARM

Base-Register Addressing: LDR R1, [R0+3] / STR R0, [R1#12]

Index Addressing: [A + R0]

Stack Addressing: Zero bit address

Intel Addressing modes فيها 23 ، موجودة

STRB R0, [R1, #12] بنزيد 12 وبثبت

هاته بتروح على الاذرى اللي بـ R1 وبعدينه بنزيد 12 وبثبت الاذرى ، يعني R1 بتضل ثابتة. offset

STRB R0, [R1, #12]! هاته بتروح على R1 رقم 12 ، بعدينه بثبت الاذرى ، يعني R1 بتتغير . Preindex

STRB R0, [R1], #12 هاته بثبت الاذرى من الميموري بعدينه بنضيف على R1 ← 10

Postindex .

بالأسمبلي ما بزبط اتعامل مع الميموري غير بال load وال store ، يعني

زي هيك ما بزبط ← ADD R0, [R1]

بتعملها هيك ← LDR R2, [R1]

ADD R0, R0, R2

Relative Address بال ARM لما بدنا نعمل برانش ، بس بنتعامل مع

Instruction length: 32 bit ← ثابت ARM ال length تاع ال

موجودة (متغير) بجد مثال. PDP-8 InstructionFormat على مثال على 24

PDP-10 " " 36

ARM " " 32

ADDS: معناها حددت قيمة ال flag بالأتوماتيك انها 1

Assembly:

* في الـ ARM في 16 رجستر: R0-R15 وجميعها حجمها bit32.

* R0-R12: General Purpose Register (يعني بقدر اعمل فيهم اي بدياه)
R13-R15: Special
R13: SP (stack pointer)
R14: LR (link Register)
R15: PC

* جدول ما بقدر اتعامل معهم واتحكم فيه زي مبدئيا لأنهم الهم وظائف محددة.

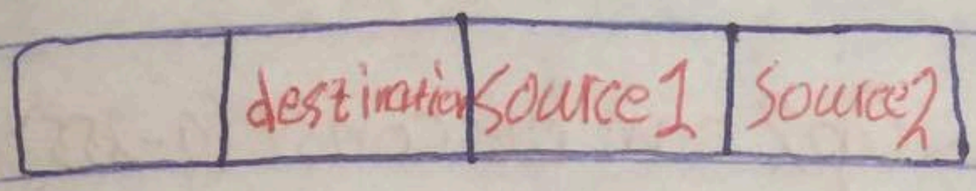
SP: Top of the stack بتأشر عل

* بما انو كل الرجسترات 32bit و معناها اصلا بقدر نتعامل مع 2^{32} من الميموري (4GB) و لأنو الـ PC رح تخزنه آدرس حجمو bit32.

* طبعاً لو بدي أأشر عل اول cell بالميموري بالـ PC: 00000000001 (32 bit)

* حجم الـ cell أيضاً 8bit (byte) داخل الميموري.

ARM Instruction Format:



Source 2 can be a register, immediate (constant), memory.

mov instruction:

* احنا بنعرف انو بال ARM ما بنقدر نتعامل مع الميموري إلا load/store طيب ايش فايدة ال mov؟

ال MOV بنستخدمها لما بدنا نتعامل مع ريجستر وريجستر أو ريجستر و const

* Mov Rn, op2 ← يعني بننقل قيمة op2 لـ Rn

op2 can be a constant num #k wich is an 8 bit value that can be 0-255 in decimal, (00-FF in hex).

Ex:

1) MOV R2, #120 ← decimal بال 120 ← ال const بنعرف # وطبعا ال 120

2) MOV R2, #0x25 ← (25) يعني ال hexa ← ال (0x) بتدل على انو الرقم بال hexa

واحنا بنعرف انو كل digit بال hexa بتمثل بـ 4bit، يعني ال 25 بال hexa، 4bit لـ digit ← 0010 0101 ← 2 5

بما انهم 8bit و R ← 32bit تلقائي بصيروا باقي ال digit يساوي اصفار ← 00000000000000100101

* لحا بدنا نكتب كومنت بنستخدم ; بدل //.

ADD instruction: → Op2 can be const (0-255) or Register.

ADD Rd, Rn, Op2 ; بجيب Rn وبعمل عليه Op2 وبخزن النتيجة في Rd

Ex: MOV R1, #0x25
MOV R7, #0x34
ADD R5, R1, R7
;(R5 = R1 + R7)

MOV R1, #0x25
OR ADD R5, R1, #0x34
;(R5 = R1 + 0x34)

R5 = 0x59 ((0x25 + 0x34) = 0x59)

SUB instruction: ; نفس الـ ADD

SUB Rd, Rn, Op2 ; بطرح OP2 من Rn (Rn - OP2) وبخزنه النتيجة في Rd

* الـ SUB والـ ADD بأثروا على الـ flags لما يجيبوا أو يطرحوا.

* بدل ARM بقدر أنا اتحكم إذا بأثروا على الـ flags أو لا عن طريق انو أضيف (S) ورا ADD أو SUB أو أي عملية بدي إياها.
إذا كانت في S ⟵ SUBS / ADDS ⟵ بكونوا بأثروا على الـ flags.

$R_1 = R_1 - 0x25$

```
SUB R1, R1, #0x25
SUB R1, #0x25
```

بعدد اكتبها هيك

$R_1 = R_1 - R_2$

```
SUB R2, R2, R2
SUB , R2, R2
```

هيدول 28 فيهم Instruction مع شرحها.

The ARM memory map:

The R13 is set aside for stack pointer.
The R14 is designated as link register wich holds the return address when the cpu calls a subrotine.
The R15 is the program counter (PC).

سبيشل لكنه ليست من فئة الـ 16

The CPSR (current program status register) is used for keeping condition flags among other things. (بتحفظ الكوندیشنه فلاجز)

The Thumb: Thumb أو Normal <= ARM بال CPU تشتغلي بعد أخلي الـ

كيف يعني؟ => بالنورمال الـ ARM بتكونه طبيعية يعني حجم الرجيستر 32bit وعدد الرجيسترات 16 وهكذا، اما بال Thumb بصير يشتغل فقط على 8 رجيسترات R0-R7 والاشتركشنه بتصير 16 بدل 32bit.

* هيدا امنا بنعرف انو بالنورمال بنقدر نتعامل مع 4GB من الميموري.

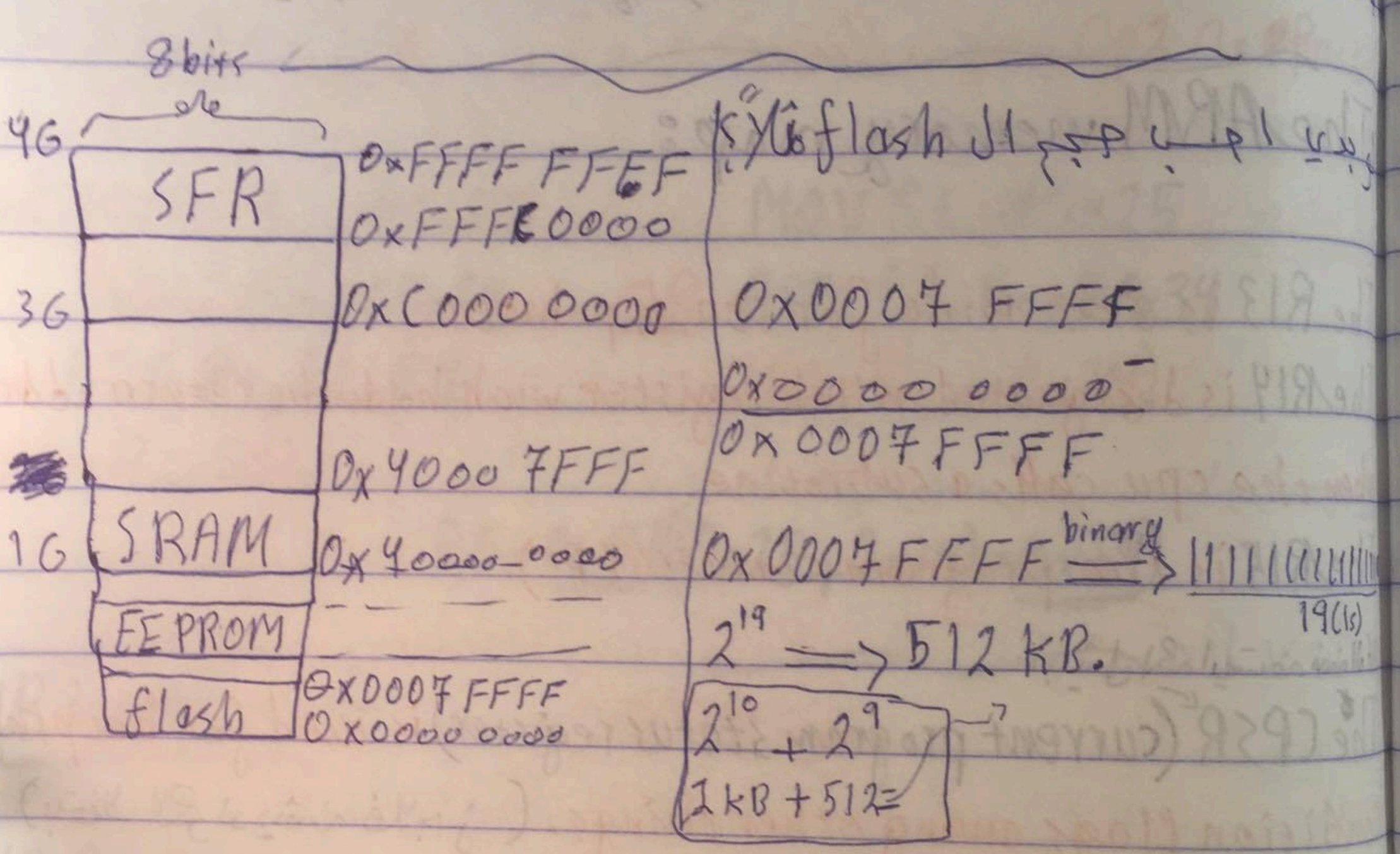
4GB for both on chip and off chip (RAM, flash ...)

onchip: هي الميموري الداخلية اللي بتكون داخل الـ CPU

off chip: ـــ ـــ الخارجية ، ، ـــ خارج ـــ ـــ

لازم نجمع معي بعض ما يتعدى الـ 4G.

صورة 29



بما انو الريجستر 32bit وكل cell بالميموري 8bits و فلما اعمليلو 4 cells وعم يقرأ لي اول MOV R1, 0x4000 0000 ~~MOV R~~ من الـ SRAM بالميموري (حسب المثال السابق) انو الريجستر 32bit

MOV R1, 0x4000 0000 : فهنا لازم نعمل هيك

LDRB R0, [R1]

بنقط B عشان

يوقفلي بس بايت

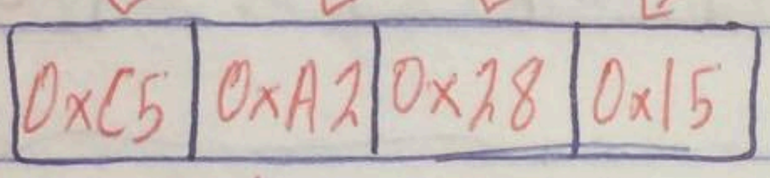
واحد

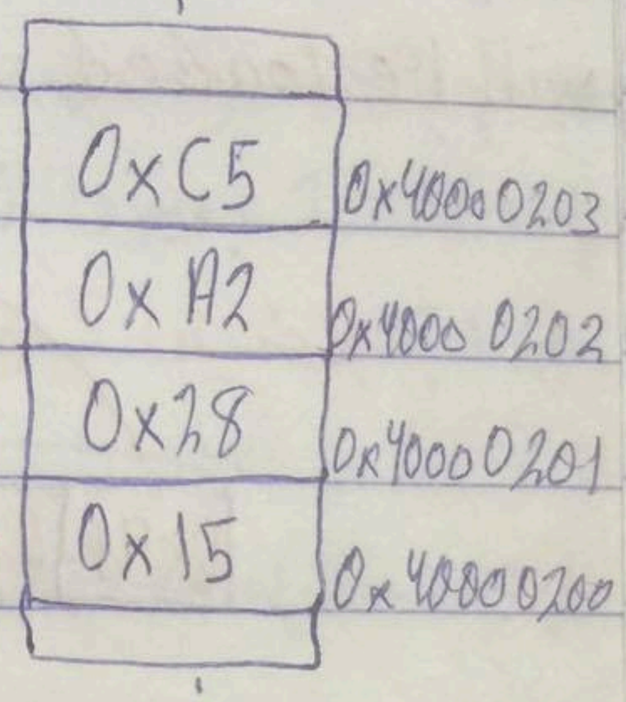
load and store instructions in ARM:

LDR Rd, [Rx] ; load Rd with the contents of location pointed.
; Rx is an address between 0x0000 0000 to 0xFFFF FFFF

Ex: assume R5 = 0x4000 0200
⟹ LDR R7, [R5]

* بما انه ما حطينا B رح يوخذ .4 cells





R7 will be loaded with: 0xC5A22815

little Indian

LDRB R7, [R5] ← * لو حطينا B

R7 will be loaded with: 0x00000015

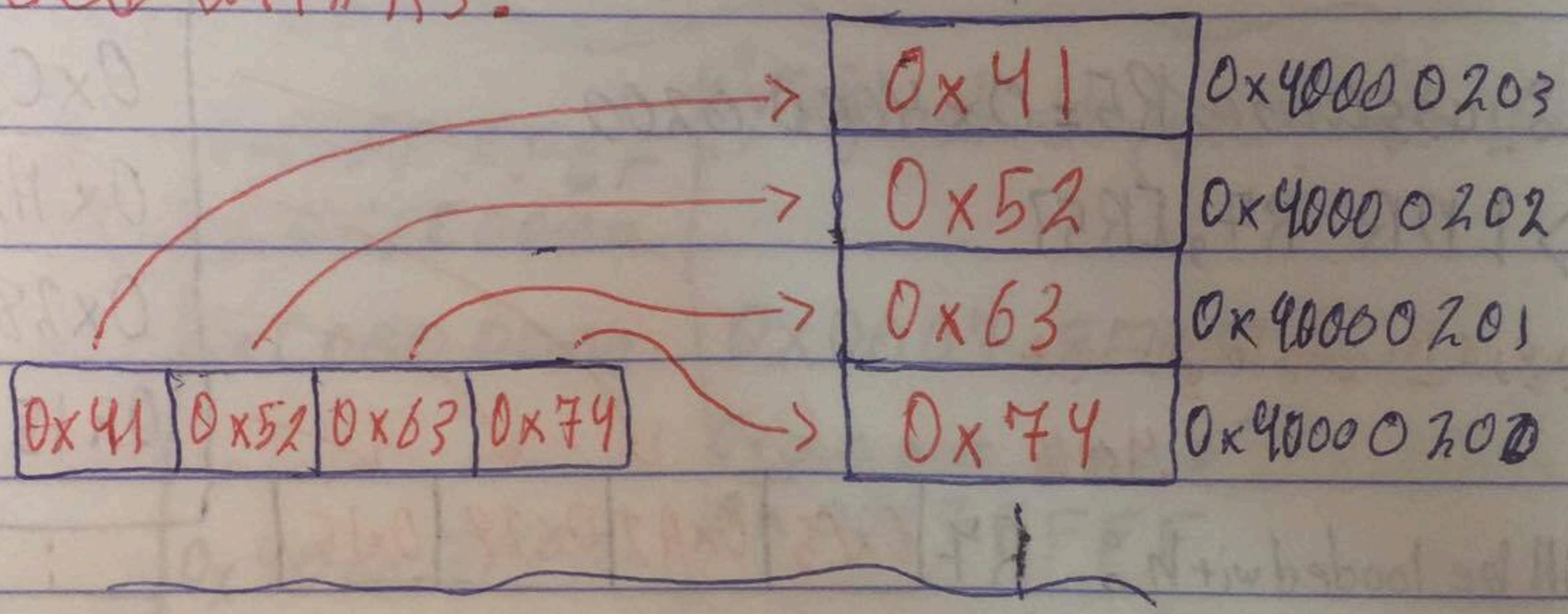
LDRH R7, [R5] ← (2 cells) Half word و يوخذ H لو حطينا *

R7 will be loaded with: 0x00002815

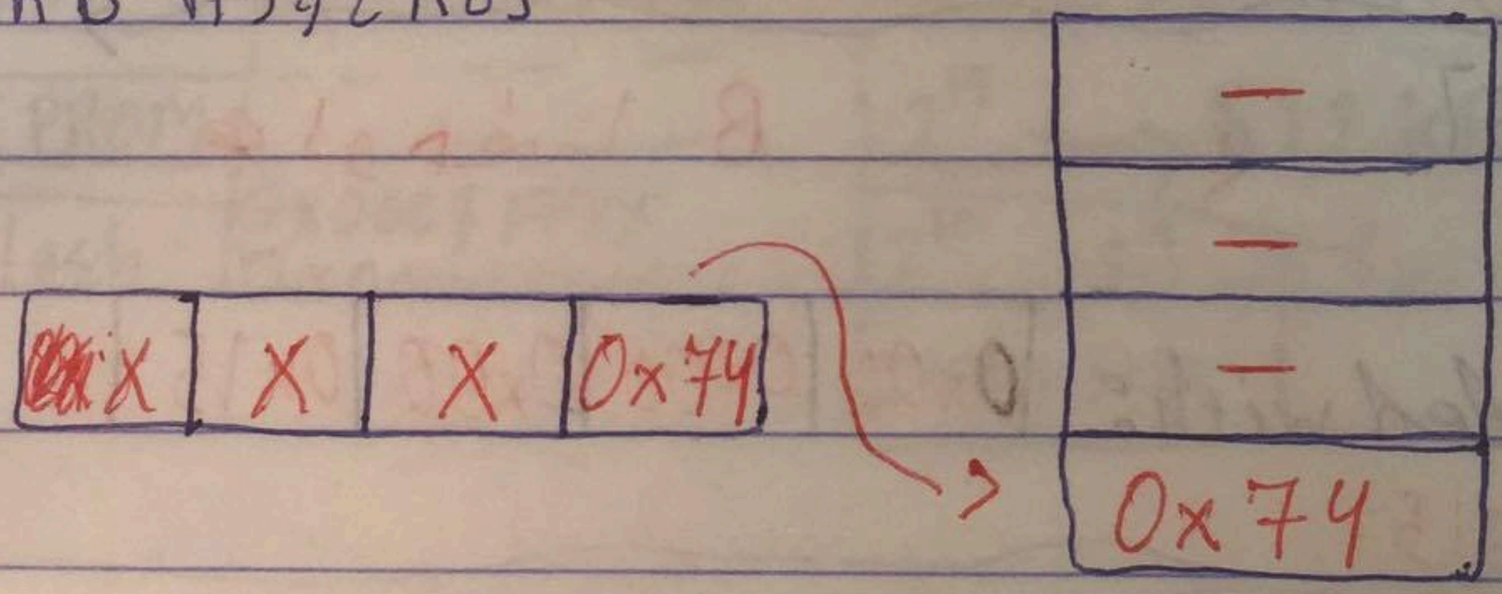
ie: STR Rx, [Rd] ; Store register Rx into address location in Rd

Ex: Assume R6 = 0x40000200 and R3 = 0x41526374.

⟹ STR R3, [R6] * location 0x40000200 through 0x40000203 will be loaded with R3.



⟹ STRB R3, [R6]



⟹ STRH R3, [R6]

* صورة 30 ، عم فرض انا عملت عملية معينة وطلع عندي carry ؟ رح
تغير 1 = C ، عم فرض طلع Over flow ؟ رح تغير 1 = V ، . . . . . .
ال T بتدل إذا ال ARM ⇐ Thump أو لا .

203

فيا لا بنستركشن فورمات تاعت ال ARM ⇐ (S) فيجو 1 bit [1 | 25]
إذا كانت 1 بصيروا ال flags يتأثروا عند العمليات ، طيب كيف بدنا التحكم
فيها ؟ بنضيف S ورا العمليات لما بدنا اياها 1 ⇐ SUBS / ADDS . . . .

Conditional Branch (Jump) ، بتعتمد عال flags .

BCS: Branch if C=1) BCC: Branch if C=0 BEQ: if Z=1

BNE: if Z=0 BMI: if N=1 BPL: if N=0

BVS: if V=1 BVC: if V=0

بنحط 0x لنعبر عن رقم بالهيكسا وما بنحط اشي لنعبر عن ديسيمال
وبنحط 2_ لنعبر عن رقم بالبايناري ⇐ MOV R6, #2_11010011
وبنحط 'char' لنعبر عن حرف ⇐ LDR R3, #'2' .

Assembler directives:

* الAssemple programming بتكونه من انستركشنز و دايركتفز.

* الdirectives بتعطي معلومات للAssembler

* صورة 31 و 32 ← انواع الدايركتفز و استخداماتها و مثال عليها.

---

* لما كنا ننقل رقم const على R كنا نستخدم MOV و بقدر تستخدم LDR أيضاً لما يكونه عندي الconst أكبر من 8bit لأنو بال MOV بس 8bit.

LDR R7, =0x112233 ← لازم نخطيها.

الdirective (EQU) بنستخدموا لحتى نعطي قيمة معينة بنستخدمها كتير تم.

الاسم القيمة

COUNT EQU 0x25

* هيدا بقدر نستخدم الاسم بدل القيمة.

MOV R2, #COUNT

الاسم الريجستر

VAL1 RN R1

VAL2 RN R2

SUM RN R3

* بقدر كمان نعطي تم لريجستر ←

* صورة 33 حيث مثال.

* الـ directive ⇐ ENTRY ، بنخطوا لما نبلش نكتب كود (Instruc
يعني أوامر.
* الـ directiv ⇐ END ، بنخطوه لما نخلص كلشي.

DCB directive (define constant byte)

* بنستخدم DCB لحتى نعرّف رقم أو string ونحجزلو مكانه بالميموري.
* هيك بنكونه حجزنا byte واحد بالميموري ⟹ MYVALUE DSB 5
في الرقم 5، MYVALUE بكونه بأشر ع الادرس ميموري اللي فيه الرقم 5.

* MYVALUE بكونه بأشر ع الميموري cell رقم 100.

100	0000 0101
101	

* لو حطينا الـ 5 هيك ⇐ '5' ، معناها رح يتخزنه الآسكي كود للرقم 5 بالميموري وليس الرقم نفسه.

* هيك بنكونه حجزنا لكل ⟹ MYMSAGE DSB "HELLOWORLD"
حرف بالجملة مكانه بالميموري حجم 1 byte وبكونو كلهم تحت بعض، وبكونه MYMSAGE بأشر ع الادرس ميموري اللي فيه أول حرف ⇐ (H).

100	0000 0101
101	H
102	E
103	L

* بما انو مسموحة بتحجز بس 1 byte، معناها اكبر رقم بقدر أخزنو هو 255.

little Indian كل اللي تمت اشتغلناه

* لو بدي أخزنه 300 معناها رح احتاج 16 bit (half Word).

DCW directive (define constant half word).

* بستخدم DCW لحتى نعرف رقم أكبر من 255 ونجزلوا مكانه بالميموري.

* لو بدي أخزنه الرقم 300 رح يخزنو بـ 2 cells بالميموري ورح يدخل Array of const

⟹ MYDATA DCW 0x20, 0xF230, 5000, 0X9CD7 hex hex dec hex

* حتى لو كانه الرقم أقل من 255 إذا حبيتو مسوونه رح يتجزء 16 bit

lowest

1000	20
1001	00

0x0020 ⟸ 16 bit 0x20

most least

* يعني لو بدي أخزنه (lowest بال least بنفس ال little Indian بال)

(بال Big // // most // // //)

صورة 56 بتوضح.

1000	20
1001	00
1002	30
1003	F2

* لو بدنا نكمل ونضيف اليه بعدو (0xF230)

* لو بدي أخزنه رقم كبير كبير جداً بستخدم 32 bit (Word)

وبستخدمها برضو لما بدنا نخزن float number.

⟹ DCD directive (define constant word)

* شونة لما بدي اخزنه الرقم رح نخزنو بـ 4 cells ← Array of const

⇒ MYDATA DCD 0x200000, 0xF30F5, 5000000

* MYDATA بكونه بلش عن أول const بال Array.

* لما بدي الـ const الثاني اليا بال Array بجوه MYDATA أربعة، لأنو كل رقم بتاخو 4byte يعني 4cells، لو بدي الرقم الثالث بجوه 8.

* بال DCW بجوه 2 عشان اجيب الرقم الثاني.

صورة 36 تالتي مغير.

* لو عندي هاد: Var1 DCB 1و2و3 ، رح يكونه مخزنه (بال هيك بالصور ي: 100 | 1 | 101 | 2 | 102 | 3 | ورح يكونه Var1 بلش عن 100. (بال

* لو بدي اجيب 1 مو 2، لازم بالأول اعلم الـ load كل واحد عن ريجستر بعدينه اعمل العملية اليا بدي اياها.

اما هيك [100]و LDRB R0 أو هيك ← LDRB R0,[Var1]

⇒ الطريقتانه صحيتانه بلكي سوف نستخدم الطريقة الثانية لأنها فعلياً لا تعرف الأدرس ميموري للعنصر وإنما هو مخزنه داخل (Var1).

LDRB Rn,[Rm] ⟸ آدرس حافظه‌ای که در Rm است را بخوان و در Rn قرار بده.

* دستورالعمل‌های انتقال داده‌ها برای کار با حافظه به کار می‌روند.

* دستورالعمل‌های انتقال داده‌ها برای کار با حافظه به کار می‌روند. Rm و Rn در این دستورالعمل‌ها شماره رجیستر هستند.

STR Rn,[Rm] ⟸ مقدار رجیستر Rn را در آدرسی از حافظه که در Rm است بنویس.

LDR Rn,[Rm,#4] ⟸ مقدار رجیستر Rm را با ۴ جمع کن و سپس مقدار حافظه در آدرس جدید را در Rn قرار بده.

STR Rn,[Rm,#4] ⟸ مقدار رجیستر Rm را با ۴ جمع کن و سپس مقدار Rn را در آدرس جدید در حافظه بنویس.

LDR Rn,[Rm],#4 ⟸ مقدار حافظه در آدرس Rm را در Rn قرار بده و سپس Rm را با ۴ جمع کن.

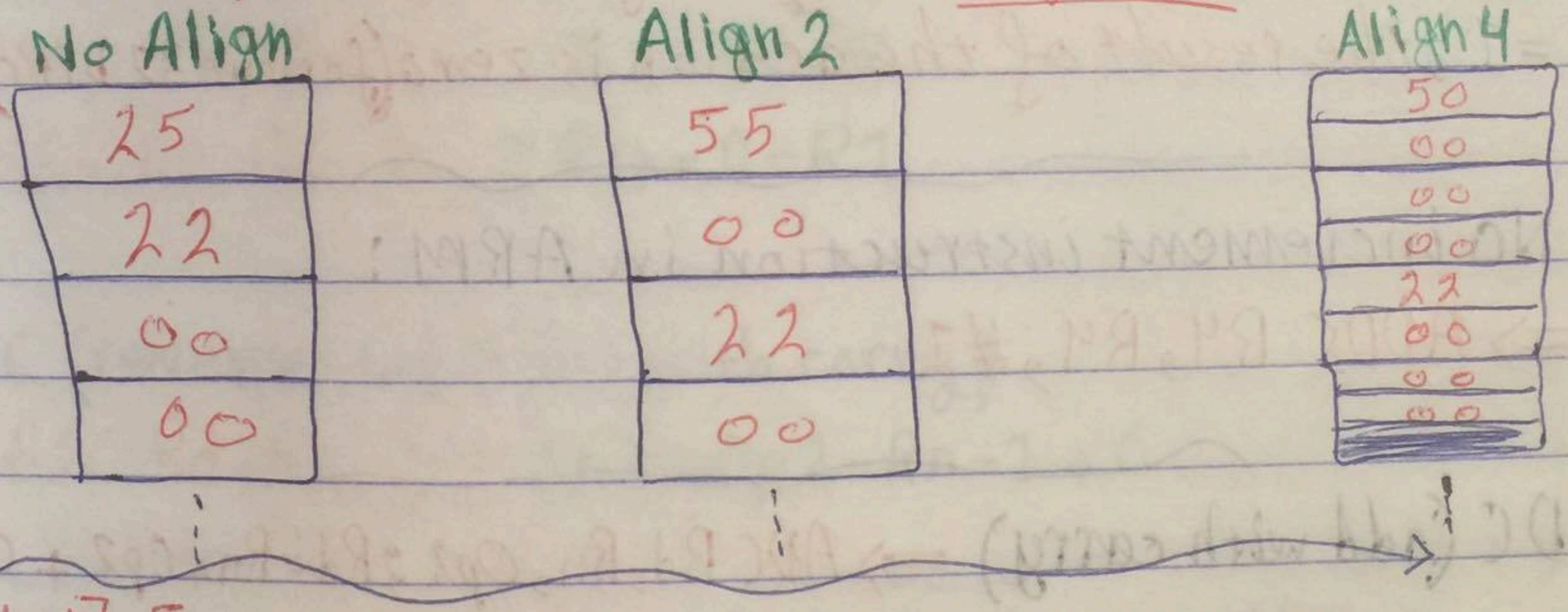
STR Rn,[Rm],#4 ⟸ مقدار Rn را در آدرس Rm در حافظه بنویس و سپس Rm را با ۴ جمع کن.

ADR directive: ADR Rn, label

فبير عندي الكود هيك: ADR R1, Var1
LDRB R0, [R1]

ALIGN: * بنستخدمها لحابدنا نجز word أو half-word.
لحتى نعطي امر للرقم ليكمل عدد البت تبعوتو.

⇒ALIGN 4; the next instruction is word (4 bytes) aligned.
⇒ALIGN 2; the next instruction is half-word (2 bytes) aligned



[label] [Instruction] [; comment]

* بقدر أضيف label للأنستركشن و هو هم بترجع الآدرس تاع الأنستركشن اللي بتحطوا عليت.

* يعني لو بدك تعمل برانش ع هاد الأنستركشن لازم تحطولا label

* الكومنت والليبل اختيارات حسب الضرورة.

Instruction format ، 35 صورة

ARM Addressing Modes:

1) Register 2) Immediate 3) Register indirect (STR/LDR)

* فقط هدول بال ARM.

Ex: LDR R2, =0xFFFFFFF5
MOV R3, #0x0B
ADDS R1, R2, R3 ; R1 = R2 + R3 and update the flags.
C = 1, since there is a carry out from D31
Z = 1, the result of the action is zero (for the 32 bits)

* No increment instruction in ARM:
⟹ ADDS R4, R4, #1

ADC (add with carry) ⟹ ADC Rd, Rn, Op2 ; Rd = Rn + Op2 + C

Subtraction of unsigned numbers

⟹ SUB Rd, Rn, Op2 ; Rd = Rn - Op2

MOV R1, #0x4C ; R1 = 0x4C
MOV R2, #0x6E ; R2 = 0x6E
Ex: SUBS R0, R1, R2 ; R0 = R1 - R2

SBC (subtract with borrow)
⇒ SBC Rd, Rn, Op2 ; Rd = Rn - Op2 - 1 + C
. Reg Const Reg

No decrement instruction in ARM
⇒ SUB R0, R0, #1

RSB (reverse subtract)
* RSB Rd, Rn, Op2 ; Rd = Op2 - Rn
⇒ MOV R1, #0x6E ; R1 = 0x6E
RSB R0, R1, #0 ; R0 = 0 - R1

RSC (reverse subtract with carry)
⇒ RSC Rd, Rn, Op2 ; Rd = Op2 - Rn - 1 + C

Multiplication of unsigned numbers in ARM:

Instruction	Source1	Source2	Destination	Result
MUL	Rn	Op2	Rd (32 bits)	Rd = Rn × Op2
UMULL	Rn	Op2	RdLo, RdHi (64 bits)	RdLo (Low) : RdHi (High) = Rn × Op2

* 8 تو الريجسترات 32 bit و فيلزمني ثنين.

بتخزنه برجيستر واحد

* لما بدي اضرب رقمين والنتيجة تكون كبيرة بنستخدم MUL 32bit لحتى
* ////// والنتيجة تكون كبيرة بنستخدم UMULL لحتى 64bit.

MUL (Multiply) ; MUL Rd, Rn, Op2 ; Rd = Rn x Op2

Ex: MOV R1, #0x25 (8bit)
MOV R2, #0x65 (8bit)
MUL R3, R1, R2 ; R3 = R1 x R2 = 0x65 x 0x25

حاصل ضربهم أقل من 32bit

Ex: LDR R1, #=100000 (8<)
LDR R2, =150000 (8<)
MUL R3, R2, R1

* رح يطلع الناتج 15000000000 وهو رقم كبير ما بزبط الزتو برجستر واحد، لهيك لازم نستخدم UMULL

UMULL (unsigned multiply long)
* UMULL RdLo, RdHi, Rn, Op2 ; RdHi:RdLo Rd = Rn x Op2

⇒ LDR R1, =0x54000000
LDR R2, =0x10000002
UMULL R3, R4, R2, R1 ; 0x54000000 x 0x10000002
= 0x0540000000A8000000
; R3 = 0xA8000000 ; the lower 32bits.
; R4 = 0x05400000 ; the higher 32bits.

Multiply and Accumulate Instructions in ARM:

⇒ MLA Rd, Rm, Rs, Rn ; Rd = Rm × Rs + Rn

* بستخدم MLA لما بدنا نضرب و نجمع مع بعض.

Ex: MOV R1, #10

MOV R2, #5

MLA R4, R1, R2, R3 ; R4 = R1 × R2 + R3 = 540

MOV R3, #40

* لما تكون النتيجة أكبر من 32 bit، بستخدم: UMLAL

UMLAL RdLo, RdHi, Rn, Op2 ; RdHi:RdLo = Rn × Op2 + RdHi:RdLo

Logic Instructions: AND: ANDing ORR: ORRing

EOR: Exclusive-ORing BIC: Bit clearing

* بروح بنعمل فلاج (S) إذا بدنا نغير الـ flags.

AND Rd, Rn, Op2 ; Rd = Rn ANDed Op2

ORR Rd, Rn, Op2 ; Rd = Rn ORed Op2

EOR Rd, Rn, Op2 ; Rd = Rn Ex-ORed with Op2

BIC (bit clear):

* BIC Rd, Rn, Op2 ; Clear certain bits of Rn specified by the Op2 and place the result in Rd.

i.e. Rd = Rn AND NOT Op2

* أي bit بدي اعملها مسح بعملها AND مع صفر فبتصير صفر.

MVN (move negative):

* MVN Rd, Rn ; move negative of Rn to Rd

; Rd على Rn بدي بنقل الـ complement و

⟹ "MVN R2, #0" will make R2 = 0xFFFFFFFF

⟹ LDR Rd, = 0xFFFFFFFF * بقدر اعملها هيك كمان ما بتفرق

Ex: LDR R2, = 0xAAAA AAAA ; R2 = 0xAAAA AAAA

MVN R0, #0 ; R0 = 0xFFFFFFFF

EOR R2, R2, R0 ; R2 = R2 ExORed with 0xFFFF FFFF ; = 0x55555555

$X \text{ xor } 0 = X$ ، $X \text{ xor } 1 = X'$

* عشان هيك إذا بدي اعكس رقم بعملو xor مع 1

* لو عندي 100 وبدي اعكس بس الـ bit الثانية ← الـ bit الثانية [2]

100
xor
100

000

بنعمل لل bit الثانية وقط xor مع 1.

```
mov R0, #4
EOR R2, R1, R0
```

* بنفس نستخدم الـ AND والـ OR بالنفي: $X AND 0 = 0 / X AND 1 = X$

$X OR 0 = X / X OR 1 = 1$

Rotate and Barrel shifter:

⇒ Barrel shifter

الـ C بتخزن قيمة آخر bit بطلع.

logical shift right — most bit — lowest bit

LSR: 0 → [MSB ⟶ LSB] ⟶ [C]

Ex: MOV R0, #0x9A
MOV R2, #0x03
MOV R1, R0, LSR R2 ; shift R0 to right R2 times and move the result to R1.

Ex: MOV R0, #0x9A
MOVS R1, R0, LSR #3 ; shift R0 to right 3 times, and then move the result to R1.

الرقم مجمو 32.

* طبعاً ال shift times منش لازم يتعدى ال 32 لأنو الرقم رح يصير كلو أصفار.
* لحابدي آخر الريجستر بعملو xor مع نفسو أو بعملو AND مع 0.

LSL: C ← MSB ← LSB ← 0

Ex: LDR R1, =0x0F000006
MOVS R2, R1, LSL #8

* طبعاً نفس الـ LSR ما بختلف.
* في المثال عملنا shift لليسار بمقدار 8 بت، يلعني زي كأني ضربت الرقم بـ 2^8.
* وبنقدر نستخدم ال left shift بعمليات الضرب وهي أسرع من MUL
* وبنقدر نستخدم ال right shift بعمليات القسمة.

Ex: TIMES EQU 0x5
LDR R1, #0x7 → ثابت — حاي المفروض بال MOV نعملها لأنو رقم صحيح
MOV R2, #TIMES
MOV R1, R1, LSL R2 ; shift R1 left R2 times and place the result in R1.

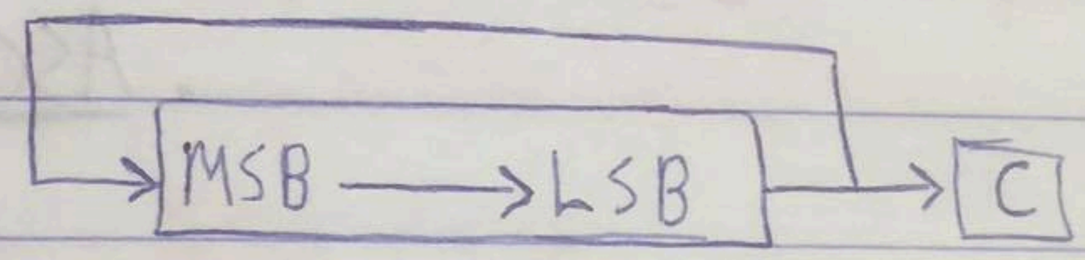
ASL

Arithmetic shift left: زي الـ logical shift left بس إذا تغيرت* في الـ most bit قيمة الـ over flow يكونة في.

ASR

Arithmetic shift right: زي الـ shift right عادي* لما نعمل shift right على الرقم يستمر over flow عادي ولما الرقم يستمر مستمر يطلع عنا ناتج عادي ويزم تخاوف على قيمة الـ sign bit.

ROR (rotate right):



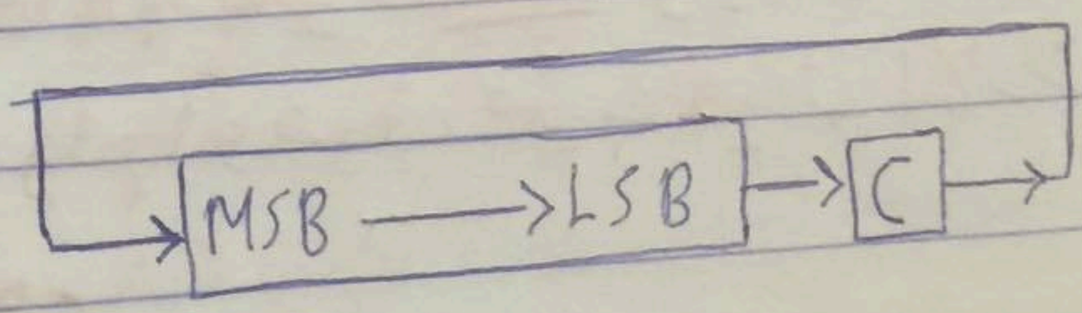
rotate left: في بعض الـ ARMs ما في rotate left فبستخدم* الـ rotate right لحتى نعمل rotate shift.

32-5 = 27

* يعني مثلاً "لو بدي اعمل 27 times بتعمل rotate left 5 times أو بدي اعمل rotate left 5 times بتعمل rotate right.

RRX (rotate right through carry):

* هيدا شغلة ثنائية مع تختلف معي بأول rotate bit لأني رح آخذ معامنة الـ C.



BCD and ASCII conversions:

1) Unpacked BCD:

* يوجد نوعان من الBCD
من 0-9

* كل digit في ال Unpacked بتمثلها بـ 8 bit.
=> يعني مثلاً الرقم 9 هيك بتمثلوا: 0000 1001

ليش صوا الأرقام من 0-9 بخزنهم 4bit، الليش بنستخدم 8bit؟
عشان نقدر نحولهم لـ ASCII.

* الأرقام من 0-9 ممكن يكونوا أرقام عادي وممكن يكونوا character
=> أي شي انا بدي اطبعوا عالشاشة لازم يكونه character.

* يعني أنا لما بدي اعمي الطبع الرقم ما بطبعو كرقم، بطبع الشكل تاعو.
ونفس الأشي بالأحرف بطبع الشكل تاع الحرف ما بطبع الحرف نفسو
لأنو الكمبيوتر ما بفهم غير اوs.

=> يعني أنا لما بدي أدخل رقم أو حرف بدخلو كـ ASCII ويعدينه بحولو لقيمتو الاسكية، ولما بدي اطبع رقم أو حرف بجيب قيمتو الاسكية وبحولو لـ ASCII بعدينه بطبعو.

* صورة 37 فيها جدول للأرقام من 0-9.

=> الأرقام والحروف كل رقم من 0-9 وكل حرف بنتلوا بـ 8bit وبدي اعولو ASCII

* لو كانة عندي رقم أكبر من 9 بصير 2 digit، فلازم افصل كل digit لحالو وأعولها Unpacked (يعني 8bit)

* هسا بعد مخول الرقم لـ 8bit هسو رح يخل رقم عادي زي ما هو بس بصير 8bit عشان أعرف اعولو ASCII. 0101 ← 0101 0000
نفس الأشي

* عشان فرقها عندي الرقم 9 (1001) وبدي اطبعوا عالشاشة، بعولو لـ 8bit ⇐ 0000 1001 بعدين بجمعلو 30H (من الجدول صورة 37) فبصير هيك 0011 1001 وهاد بمثل شكل الرقم 9.

* لو بدنا نقرأ الرقم 9 رح نقراو هيك 0011 1001 فلازم نطرح 30H لحتى يصير Unpacked 0000 1001.

* يعني لما بدي اقرأ أو أطبع رح استخدم الـ Unpacked، لما يكون الرقم أعلا بقدر امثلو بـ 4bit فرح يأخذ مساحة عشان هيك لما بدنا نخزنوا بالصور بنستخدم النوع الثاني اللي هو Packed BCD

2) Packed BCD:

* بتمثلي كل digit من 0-9 بـ 4bit.
* يعني بعتبر بالبايت الواحد أخزنه رقمين.
* مثلاً لو كانه عندي رقم أكبر من 9 مثلاً 18 ما بتمثلوا هيك 18 هيك لازم 0001 1000.
BCD

* إما حكينا انو لحابنا نقرأ أو نعرض حالتة بنستخدم Unpacked لـ 18 لازم نقسما لقسمين ونعرض كل digit لحال 0001 1000.
1 2

=> أول شي بنعرض الواحد بعدين الثنين عشان يطلع 18 فبنيب ال 0001 1000 وبنعملو 4bit right shift => 0000 0001
بعدين لازم اجعلو 30H => 0011 0001 بعدين بنعرضوا.

* هيك لازم نعرض الثنين 8 بنيب ال 1000 0000 وبنعملو AND مع 1111 0000 => 0000 1000 بعدين لازم اجعلو 30H
0011 1000 بعدين بنعرضوا.

Ex: ASCII to packed BCD conversion:

1) MOV R0, #0x29
2) ANd R1, R0, #0x0F ; mask upper four bit's.
3) ORR R1, R1, #0x30 ; combine with 30 to get ASCII
4) MOV R2, R0, LSR #4 ; shift right 4 bit's to get Unpacked BCD
5) ORR R2, R2, #0x30 ; combine with 30 to get ASCII

*شرح المثال:

1) R0 = 0010 1001. 2) 0010 1001 AND 0000 1111

⇒ 0000 1001 = R1

3) 0000 1001 OR 0011 0000 ⇒ 0011 1001 = R1

*هيك بتكون R1 جاهزة للإباعة.

4) 0010 1001 ⟶ 0000 0010 = R2

5) 0000 0010 ORR 0011 0000 ⇒ 0011 0010 = R2

*هيك بتكون R2 جاهزة للإباعة.

Ex: ASCII to packed BCD conversion:

⇒1) MOV R1, #0x37

2) MOV R2, #0x32

3) AND R1, R1, #0x0F ; mask 3 to get unpacked BCD

4) AND R2, R2, #0x0F ;

5) MOV R3, R2, LSL #4 ; shift R2 4 bit's to left to get R3 = 0x20

6) ORR R2, R3, R1 ; OR them to get packed BCD, R4 = 0x27.

*شرح المثال:

1) R1 = 0011 0111 ⟩ 2) R2 = 0011 0010

3) 0011 0111 AND 0000 1111

⇒ 0000 0111 = R1

4) 0011 0010 AND 0000 1111

⇒ 0000 0010 = R2

5) 0010 0000 = R3

6) 0010 0000 ORR 0000 0111

= 0010 0111 = R2

chapter 4: Branch, call, and looping in ARM

* صورة 38 فيها أمثلة و 39. * صورة 40 فيها أنواع البرانش مع أمثلة.

Comparison of unsigned numbers:

⟹ CMP Rn, Op2 ; compare Rn with Op2 and set the flags

if Rn > Op2 : C=1, Z=0
if Rn = Op2 : C=1, Z=1
if Rn < Op2 : C=0, Z=0

* صورة 41 فيها أمثلة.

⟹ TST Rn, Op2 ; Rn AND with Op2 and flag bits are Update

* هي نفس الـ AND بس بتختلف بإنو انا ما بتابع أخزنه نتيجة الـ AND.
يعني بس بفحص نتيجة الـ AND وبأثر على الـ flags بدون ما أخزنه نتيجة الـ AND.

* بنستخدم الـ TST لما بدنا نفحص قيمة bit معينة.

⇐ يعني لو بدنا نفحص قيمة الـ bit الثالثة 1011 0011 بنعملها TST مع هاد 0000 0100 وبكرينه بنشوف الـ Z flag إذا قيمتها واحد معناتا قيمة الـ bit الثالثة كانت صفر والعكس صحيح.

TEQ (test equal)

* => TEQ Rn, Op2 ; Rn EX-ORed with op2 and flag bit's are set.

* بنستخدمها لما بدنا نعرف إذا قيمتين متساويات أو لا.

* صورة 42 فيها أمثلة.

B (Branch):

* بنستخدمها لما بدنا نعمل Jump بدون شرط.

* صورة 43 فيها أمثلة.

Here B Here

Here BAL Here

(Branch always)

* بنستخدمها لما نخلي البرنامج عالق يفل واقف مكانو.

نفس

B L (Branch and link) instruction and calling subroutine:

* لما بدنا نستدعي subroutine ونرجع نكمل بنستخدم BL لأنو الـ BL بتخزن قيمة الـ PC داخل R14 (LR) عشان بعد ما نخلص تنفيذ الـ sub نرجع نكمل عمل ما وقفنا.

* هاي بتختلف بال sub <= BX LR عشان نرجع نكمل عمل موقفنا.

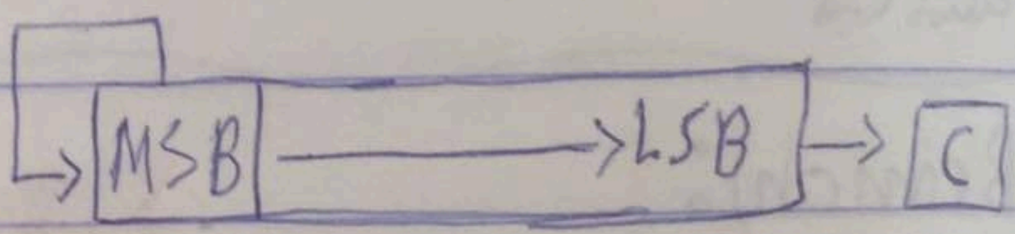
* صورة 44 فيها مثال.

* صورة 45 فيها أنواع الـ Branch الـ signed number و 8 أنستركشن تاع الفرق الـ signed number و 8 أنستركشن تاع المقارنة برضو.

Arithmetic shift: * Arithmeric shift الـ Signed number لـ

* Arithmtc right shift هو الـ بختلف عن الـ logic الـ left, right

ASR (arithmetic shift right):



MOV Rn, Op2, ASR count

=> MOV R0, #-10 ; R0=-10 = 0xFFFF FFF6

MOV R3, R0, ASR #1 ; R0 is arthmtc shifted right once

; R3= 0xFFFFFFFB = -5

الكود بلغة سي أو جافا هو High level language، الكمبيوتر ما بيتعامل غير مع واحد وصفر، عشانه هيك ما بيفهم غير لغة Assembly، فال compiler بيحول من High level لـ Assembly، بعدين بيجي Assembler بيحول من Assembly لـ Binary machine language (لغة الكمبيوتر).

C language ← Compiler ← Assembly ← Assembler ← Binary langu
00101111100
10001111110

The computer consist of:

1) CPU (central Prosser unit):

ال CPU تقوم بالعمليات الحسابية (جمع/طرح/ضرب/قسمة) وتقوم أيضاً بال logical operation (AND/OR) وتقوم أيضاً بعمليات ال Shift (Right/left) وتقوم أيضاً بعمليات نقل المعلومات بينها وبينه أجزاء الكمبيوتر الأخرى.

2) Main memory:

Memory System هي تعتبر المخزنة للمعلومات والكود والمقصود بها كل ال الي بتكونه منها الكمبيوتر.

3) I/O (Input/ouputs):

بكونه مشبوك عليه الطابعة والماوس والكيبورد و الشاشة الخ....

4) System Bus (ناقل):

تقوم بنقل المعلومات بينه أجزاء الكمبيوتر.

The CPU consist of:

1) Registers:

هي memory صغيرة و تقوم الـ CPU بتخزينه المعلومات فيتم بشكل مؤقت يعني لو بدي أجمع رقمين و بخزنة الرقمين بشكل مؤقت بعدين يقوم بالعمليات.

2) ALU:

هي المسؤولة عن جميع العمليات الحسابية بالاضافة إلى logic operation

3) Control Unit:

هي تقوم بعملية التحكم داخل الـ CPU و يعني هي مثلاً اللي بتقول للـ ALU شو يشتغل جمع و طرح و الخ... و هي اللي بتحدد أي رجسترات بدي أجمع و رجستر1 مع رجستر2 و لا رجستر؟ مع رجستر3 الخ...

4) Internal Bus:

نقل المعلومات بين أجزاء الـ CPU.

* توجد الـ Cash أيضاً داخل CPU. (حديثاً صاروا يحطوها)

Cache memory:

* توجد بين CPU و Main memory.
* هي أسرع و أكبر من Main memory.
* يتم فيها تخزين المعلومات التي استخدمناها حديثاً و عشان اذا بدنا نرجع نستخدمها نجيبها من الـ cache بدل من الـ main memory (أسرع).