

Overview of Computer Architecture

Assembly Programming

Carl Henrik Ek - carlhenrik.ek@bristol.ac.uk

November 11, 2019

<http://carlhenrik.com>

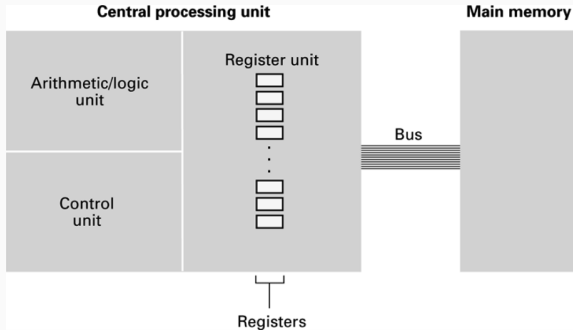
Today

- Re-cap of last time
- Assembly programming
- Preparation for lab

Execution



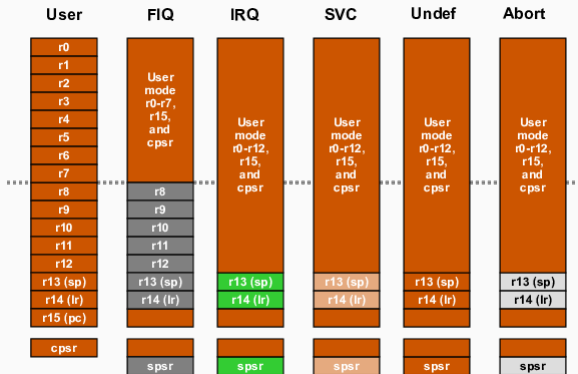
Computer Architecture



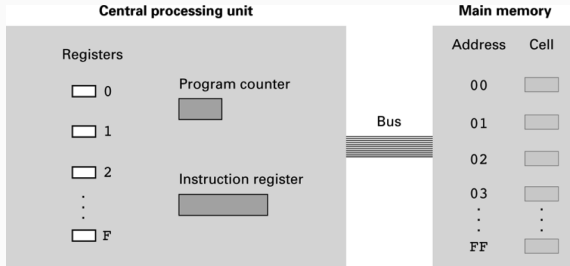
RAM as an array of bytes

Content:	FF	00	57	92	B3	8A	10	46	DC
Address:	000 000 000	000 000 001	000 000 002	000 000 003	000 000 004	000 000 005	...	134 217 725	134 217 726	134 217 727

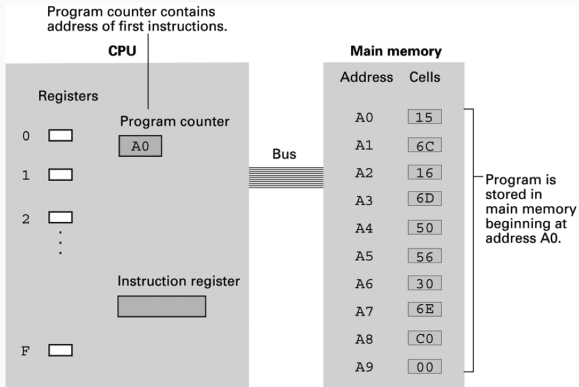
Register



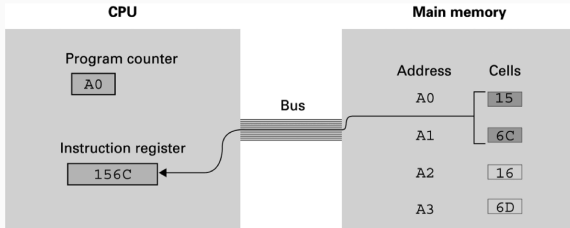
Execution



Execution

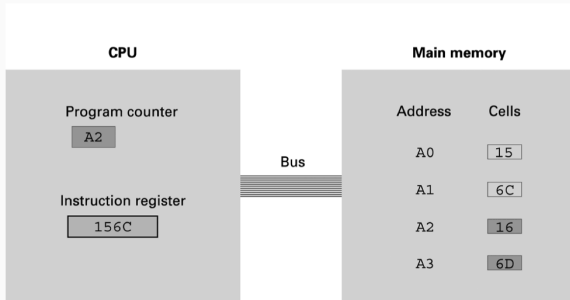


Execution



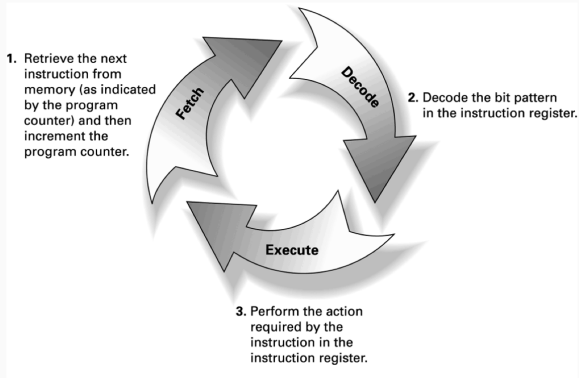
- a. At the beginning of the fetch step the instruction starting at address A0 is retrieved from memory and placed in the instruction register.

Execution

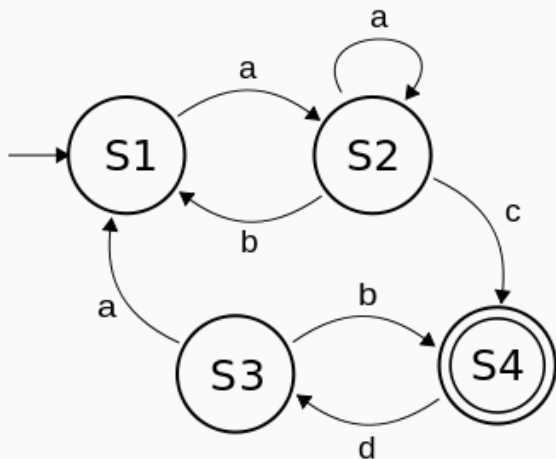


b. Then the program counter is incremented so that it points to the next instruction.

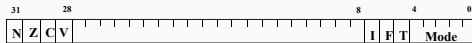
Execution



State Machine



Status Register



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

- * **Condition Code Flags**

N = **N**egative result from ALU flag.

Z = **Z**ero result from ALU flag.

C = ALU operation **C**arried out

V = ALU operation **oV**erflowed

- * **Mode Bits**

M[4:0] define the processor mode.

- * **Interrupt Disable bits.**

I = 1, disables the IRQ.

F = 1, disables the FIQ.

- * **T Bit (Architecture v4T only)**

T = 0, Processor in ARM state

T = 1, Processor in Thumb state

Flags

	Logical Instruction	Arithmetic Instruction
<u>Flag</u>		
Negative (N='1')	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z='1')	Result is all zeroes	Result of operation was zero
Carry (C='1')	After Shift operation '1' was left in carry flag	Result was greater than 32 bits
oVerflow (V='1')	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

Conditions

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition			0	0	1	OPCODE					S	Rn			Rs			OPERAND-2												Data processing		

0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI - N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear, or N clear and V set (<, or =)

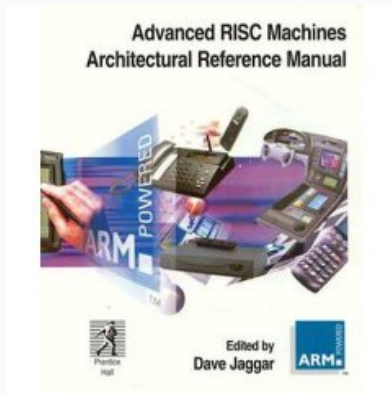
1110 = AL - always

1111 = NV - reserved.

- OP-Code
 - Decides operation
- Operand
 - OP-Code dependent
 - "Parameters"
- Each instruction on ARM 32 bits

Instruction ARM

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	Instruction Type											
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2								Data processing															
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply											
Condition				0	0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs				1	0	0	1	Rm				Long Multiply										
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap											
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET								Load/Store - Byte/Word															
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST								Load/Store Multiple																			
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off											
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off											
Condition				1	0	1	L	BRANCH OFFSET																												Branch							
Condition				0	0	0	1	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange									
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum				OFFSET								COPROCESSOR DATA XFER											
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum				OP-2				0	CRm				COPROCESSOR DATA OP										
Condition								OP-1				L	CRn				Rd				CPNum				OP-2				1	CRm				COPROCESSOR REG XFER									
Condition				1	1	1	1	SWI NUMBER																												Software Interrupt							



- Data Transfer
 - LDR, STR
- Flow Control
 - B
- Arithmetic
 - ADD, SUB, MUL

`ldr r0,[r1,#4]` Load word addressed by $R1+4$.

`str r0,[r1],#4` Store R0 to word addressed by R1. Increment R1 by 4.

`ldr r0,[r1,#4]!` Load word addressed by $R1+4$. Increment R1 by 4.

`ldr r0,=label` Load address of label `label` into R0

ADD

Code

```
_start:
```

```
    add    r0, r0, r1
```

```
    add    r0, #4
```

The Pipeline



Freedom 0 The freedom to run the program, for any purpose.

Freedom 0 The freedom to run the program, for any purpose.

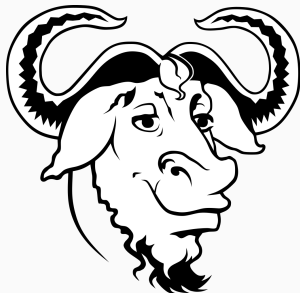
Freedom 1 The freedom to study how the program works, and change it so it does your computing as you wish.
Access to the source code is a precondition for this.

- Freedom 0** The freedom to run the program, for any purpose.
- Freedom 1** The freedom to study how the program works, and change it so it does your computing as you wish.
Access to the source code is a precondition for this.
- Freedom 2** The freedom to redistribute copies so you can help your neighbor.

- Freedom 0** The freedom to run the program, for any purpose.
- Freedom 1** The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.
- Freedom 2** The freedom to redistribute copies so you can help your neighbor.
- Freedom 3** The freedom to distribute copies of your modified versions to others. By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this

GNU/Linux

- GNU - GNU is Not Unix
- Linux defines the Kernel
 - small but very critical
 - developed with GNU tools
 - GNU Hurd - kernel
- Most other things are GNU
 - Development tools
 - Editor (Emacs)
 - GNOME







- Write code
- Assemble code to object file
- Link object files to resolve reference
- Run executable

Mnemonics

- `mov r0, #42`
- `eor r1, r1, r1`
- `add r1, r1, r0`
- `b _start`

Machine Code

- `0xe3a0002a`
- `0xe0211001`
- `0xe0811000`
- `0xeaffffffb`

Code

```
arm-none-eabi-as -o <object>.o <code>.s  
arm-none-eabi-ld -o <executable> <object1>.o
```

Assembler takes the source file and creates

Linker links several objects to a single binary

`.section` tells the assembler that what follows is a `.text` for code or `.data` for data. Each section can be placed in different locations in memory.

Directives

- `.section` tells the assembler that what follows is a `.text` for code or `.data` for data. Each section can be placed in different locations in memory.
- `.align` indicates that the following section should be aligned on 2-byte boundaries in memory.

Directives

- `.section` tells the assembler that what follows is a `.text` for code or `.data` for data. Each section can be placed in different locations in memory.
- `.align` indicates that the following section should be aligned on 2-byte boundaries in memory.
- `.global` indicates that the label `_start` should be visible outside this file. This is a directive that is used when the linker combines several different objective files in order to create a single binary

Directives

- `.section` tells the assembler that what follows is a `.text` for code or `.data` for data. Each section can be placed in different locations in memory.
- `.align` indicates that the following section should be aligned on 2-byte boundaries in memory.
- `.global` indicates that the label `_start` should be visible outside this file. This is a directive that is used when the linker combines several different objective files in order to create a single binary
- `.asciz` decides that the information that follows should be interpreted as a text and converted using the `ascii` table.

Directives

- `.section` tells the assembler that what follows is a `.text` for code or `.data` for data. Each section can be placed in different locations in memory.
- `.align` indicates that the following section should be aligned on 2-byte boundaries in memory.
- `.global` indicates that the label `_start` should be visible outside this file. This is a directive that is used when the linker combines several different objective files in order to create a single binary
- `.asciz` decides that the information that follows should be interpreted as a text and converted using the `ascii` table.
- `.word` indicates that the data that follows should be interpreted as a sequence of 32bit elements

Code

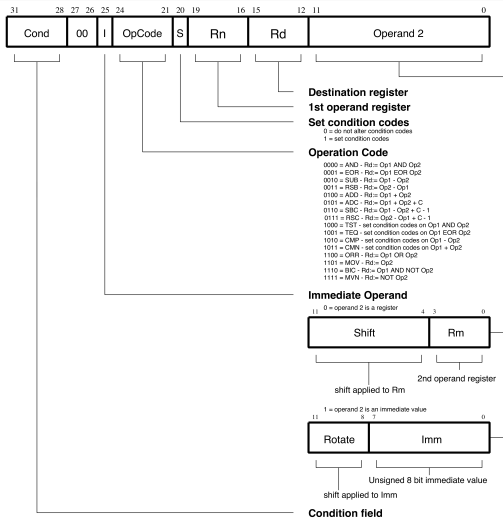
```
.section          .text
.global           _start
.align            4

_start:

    mov            r0, #42
    eor            r1, r1, r1
    add            r1, r1, r0
    b              _start
```

Code

```
0x8000:           e3a0002a
0x8004:           e0211001
0x8008:           e0811000
0x800c:           eaffffffb
```

```
mov r0, #42 @ e3a0002a
```

31-28 1110

20 0

27-26 00

19-16 0000

25 1

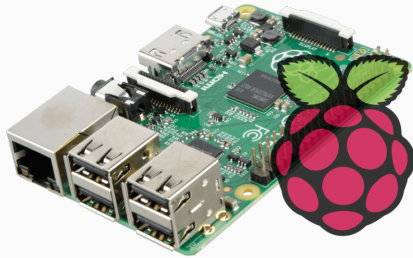
15-12 0000

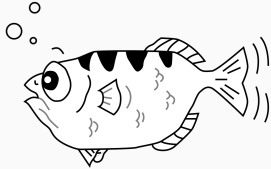
24-21 1101

11-0 000000101010

- The assembler is just a "stupid" translator, it converts mnemonics to binary
- Resolves addresses
- Re-writes code to be PC-relative
- Organises code into block

Executing Code





```

Breakpoint 1, 0x00000054 in _start ()
-----[ registers ]-----
$r0 : 0x00000000
$r1 : 0x00000000
$r2 : 0x00000000
$r3 : 0x00000000
$r4 : 0x00000000
$r5 : 0x00000000
$r6 : 0x00000000
$r7 : 0x00000000
$r8 : 0x00000000
$r9 : 0x00000000
$r10 : 0x00000000
$r11 : 0x00000000
$r12 : 0x00000000
$sp : 0xbefff850 -> 0x00000001
$lr : 0x00000000
$pc : 0x00000054 -> < start+0> push {r11, lr}
$cpsr : [thumb fast interrupt overflow carry zero negative]
-----[ stack ]-----
0xbefff850|+0x00: 0x00000001 <-$sp
0xbefff854|+0x04: 0xbefff94e -> "/home/pi/lab/gdb-example"
0xbefff858|+0x08: 0x00000000
0xbefff85c|+0x0c: 0xbefff967 -> "TERM=vt100"
0xbefff860|+0x10: 0xbefff972 -> "SHELL=/bin/bash"
0xbefff864|+0x14: 0xbefff982 -> 0x5f474458
0xbefff868|+0x18: 0xbefff9d1 -> "LC_ALL=en_US.UTF-8"
0xbefff86c|+0x1c: 0xbefff9e4 -> "USER=pi"
-----[ code:armv4t ]-----
0x803c andeq r8, r0, r0
0x8040 andeq r8, r0, r0
0x8044 muleq r0, r4, r0
0x8048 muleq r0, r4, r0
0x804c andeq r0, r0, r5
0x8050 andeq r0, r0, r0
-> 0x8054 < start+0> push {r11, lr}
0x8058 < start+4> add r11, sp, #0
0x805c < start+8> sub sp, sp, #16
0x8060 < start+12> mov r0, #1
0x8064 < start+16> mov r1, #2
0x8068 < start+20> bl 0x8074 <max>
-----[ threads ]-----
[#0] Id 1, Name: "gdb-example", stopped, reason: BREAKPOINT
-----[ trace ]-----
[#0] 0x8054->Name: start()
-----
gef>

```

Assembler Programming

Code

```
.....  
bl      _sub    @ call _sub  
.....    @ will return here  
  
_sub:  
stmdb   sp!, {r2-r3,r7}  
  
ldmia   sp!, {r2-r3,r7}  
mov     pc, r14
```


Loops

Code

```
        mov        r7, #42-1
_loop:
        sub        r7, r7, #1
        bne        _loop
```

Register to register `MOV R0, R1`

Register to register `MOV R0, R1`

Absolute `LDR R0, MEM`

Register to register `MOV R0, R1`

Absolute `LDR R0, MEM`

Literal `MOV R0, #15`

Register to register `MOV R0, R1`

Absolute `LDR R0, MEM`

Literal `MOV R0, #15`

Indexed, base `LDR R0, [R1]`

Register to register `MOV R0, R1`

Absolute `LDR R0, MEM`

Literal `MOV R0, #15`

Indexed, base `LDR R0, [R1]`

Pre-indexed `LDR R0, [R1, #4]`

Pre-indexed LDR R0, [R1, #4] !

Pre-indexed `LDR R0, [R1, #4]!`

Post-indexing `LDR R0, [R1], #4`

Pre-indexed LDR R0, [R1, #4]!

Post-indexing LDR R0, [R1], #4

Double Reg indirect LDR R0, [R1, R2]

Pre-indexed LDR R0, [R1, #4]!

Post-indexing LDR R0, [R1], #4

Double Reg indirect LDR R0, [R1, R2]

Double Reg indirect LDR R0, [R1, r2, LSL #2]

Pre-indexed LDR R0, [R1, #4]!

Post-indexing LDR R0, [R1], #4

Double Reg indirect LDR R0, [R1, R2]

Double Reg indirect LDR R0, [R1, r2, LSL #2]

Program counter relative LDR R0, [PC, #offset]

Code

```
.section          .text
.global          _start
.align           4

_start:
    mov          r0, #42
    add          r1, r1, r0

_loop
    subs         r0, r0, #1
    bne          [pc, #-4]

_end:
    b            [pc]
```

Examples Code

Fibonacci Sequence

$$f(n) = f(n-1) + f(n-2)$$

$$f(1) = f(2) = 1$$

Summary

Summary

- Code execution ✓
- Code creation ✓
- Toolchain ✓

Summary

- Code execution ✓
- Code creation ✓
- Toolchain ✓
- Practice makes perfect