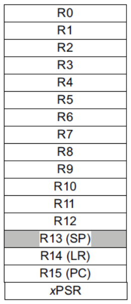
EECS 373
Design of Microprocessor-Based Systems

Prabal Dutta
University of Michigan

Lecture 2: Architecture, Assembly, and ABI
September 4, 2014

Slides developed in part by
Mark Brehob


R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
xPSR

Announcements

- Website up
  - http://www.eecs.umich.edu/~prabal/teaching/eecs373/
- Homework 1 posted (mostly a 270 review)
- Lab and office hours posted on-line.
  - **My office hours:** Tuesdays 1:30-3:00 pm in 4773 BBB
- Projects
  - Start thinking about them now!

Today…

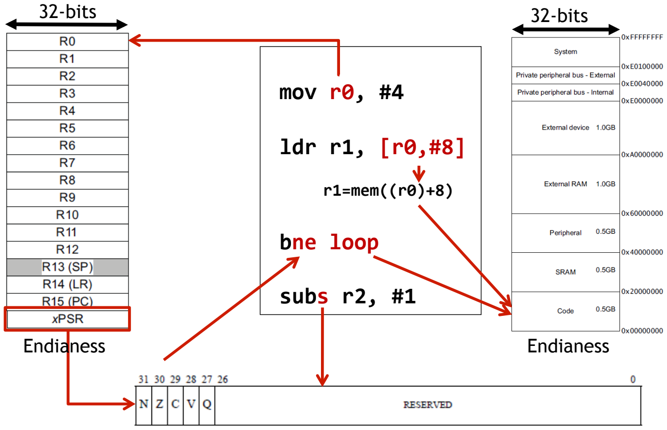
Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)

Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)



The endianess religious war: 288 years and counting!

- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war
- Historical Inspiration
  - Jonathan Swift
  - *Gulliver's Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church
    - Now a major motion picture!

- Little-Endian
  - LSB is at lower address

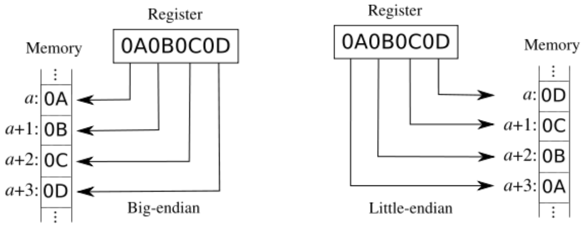
```
                               Memory     Value
                               Offset  (LSB) (MSB)
                               ======  ===========
uint8_t a  = 1;                0x0000  01 02 FF 00
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;       0x0004  78 56 34 12
```

- Big-Endian
  - MSB is at lower address

```
                               Memory     Value
                               Offset  (LSB) (MSB)
                               ======  ===========
uint8_t a  = 1;                0x0000  01 02 00 FF
uint8_t b  = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;       0x0004  12 34 56 78
```

Addressing: Big Endian vs Little Endian (370 slide)

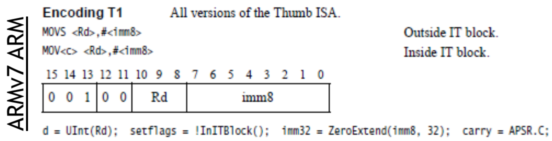
- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big - The opposite, most significant byte first
  - MIPS is big endian, x86 is little endian



Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct
- How? Refer to the "ARM ARM"

```
Instructions        Register Value             Memory Value
movs r0, #10        001|00|000|00001010        (LSB) (MSB)
                    (msb)                (lsb) 0a 20 00 21
movs r1, #0         001|00|001|00000000
```



ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

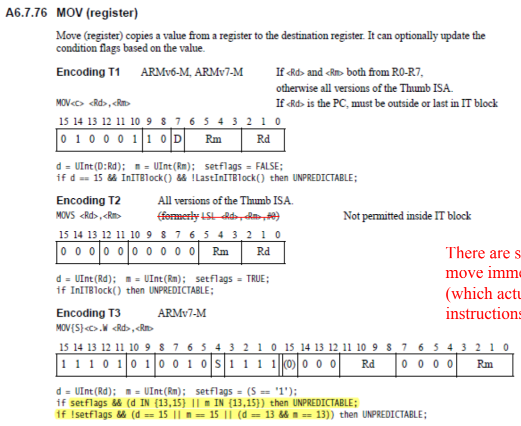
Assembly example

```
data:
    .byte 0x12, 20, 0x20, -1
func:
        mov r0, #0
        mov r4, #0
        movw   r1, #:lower16:data
        movt   r1, #:upper16:data
top:    ldrb   r2, [r1],#1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne top
```

Instructions used

- mov
  - Moves data from register or immediate.
  - Or also from shifted register or immediate!
    - the mov assembly instruction maps to a bunch of different encodings!
  - If immediate it might be a 16-bit or 32-bit instruction
    - Not all values possible
    - why?
- movw
  - Actually an alias to mov
    - "w" is "wide"
    - hints at 16-bit immediate

From the ARMv7-M Architecture Reference Manual
(posted on the website under references)



A6.7.76 MOV (register)

Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M If <Rd> and <Rm> both from R0-R7, otherwise all versions of the Thumb ISA.

MOV<c> <Rd>,<Rm> If <Rd> is the PC, must be outside or last in IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb ISA.

MOVS <Rd>,<Rm> ~~(formerly LSL <Rd>,<Rm>,#0)~~ Not permitted inside IT block

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

MOV{S}<c>.W <Rd>,<Rm>

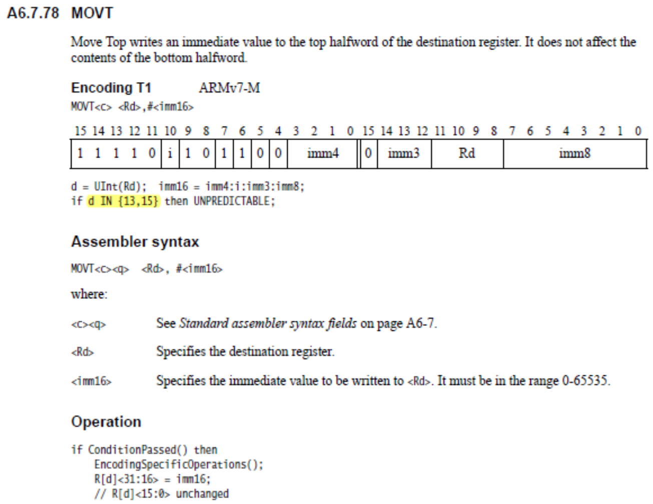
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if setflags && (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
if !setflags && (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
```

There are similar entries for move immediate, move shifted (which actually maps to different instructions) etc.

Directives

- **#:lower16:data**
  - What does that do?
  - Why?



A6.7.78 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

Encoding T1 ARMv7-M

MOVT<c> <Rd>,#<imm16>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

MOVT<c><q> <Rd>, #<imm16>

where:

<c><q> See *Standard assembler syntax fields* on page A6-7.

<Rd> Specifies the destination register.

<imm16> Specifies the immediate value to be written to <Rd>. It must be in the range 0-65535.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Loads!

- ldrb -- Load register byte
  - Note this takes an 8-bit value and moves it into a 32-bit location!
    - Zeros out the top 24 bits
- ldrsb -- Load register signed byte
  - Note this also takes an 8-bit value and moves it into a 32-bit location!
    - Uses sign extension for the top 24 bits

Addressing Modes

- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

So what does the program _do_?

```
data:
    .byte 0x12, 20, 0x20, -1
func:
        mov r0, #0
        mov r4, #0
        movw   r1, #:lower16:data
        movt   r1, #:upper16:data
top:    ldrb   r2, [r1],#1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne top
```

Today…

Finish ARM assembly example from last time

Walk though of the ARM ISA

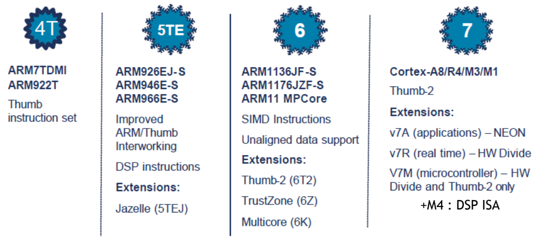
Software Development Tool Flow

Application Binary Interface (ABI)

An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- ***Calling conventions***
  - Really not part of the ISA (usually)
  - Rather part of the ABI
  - But the ISA often provides meaningful support.

ARM Architecture roadmap



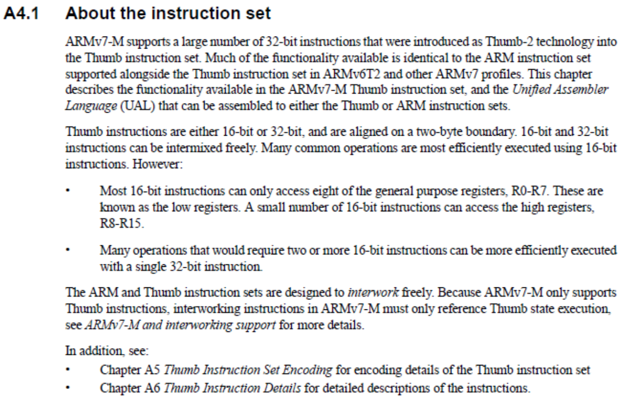
ARM7TDMI
ARM922T
Thumb instruction set

ARM926EJ-S
ARM946E-S
ARM966E-S
Improved ARM/Thumb Interworking
DSP instructions
Extensions:
Jazelle (5TEJ)

ARM1136JF-S
ARM1176JZF-S
ARM11 MPCore
SIMD Instructions
Unaligned data support
Extensions:
Thumb-2 (6T2)
TrustZone (6Z)
Multicore (6K)

Cortex-A8/R4/M3/M1
Thumb-2
Extensions:
v7A (applications) – NEON
v7R (real time) – HW Divide
V7M (microcontroller) – HW Divide and Thumb-2 only
+M4 : DSP ISA

A quick comment on the ISA:
From: ARMv7-M Architecture Reference Manual



A4.1 About the instruction set

ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. Much of the functionality available is identical to the ARM instruction set supported alongside the Thumb instruction set in ARMv6T2 and other ARMv7 profiles. This chapter describes the functionality available in the ARMv7-M Thumb instruction set, and the *Unified Assembler Language* (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R15.
- Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.

The ARM and Thumb instruction sets are designed to *interwork* freely. Because ARMv7-M only supports Thumb instructions, interworking instructions in ARMv7-M must only reference Thumb state execution, see *ARMv7-M and interworking support* for more details.

In addition, see:

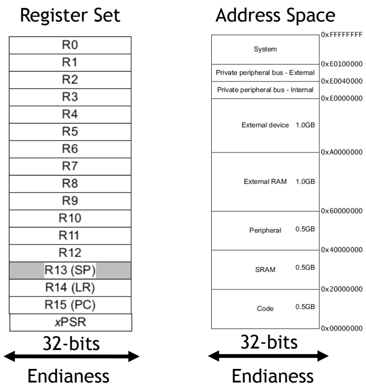
- Chapter A5 *Thumb Instruction Set Encoding* for encoding details of the Thumb instruction set
- Chapter A6 *Thumb Instruction Details* for detailed descriptions of the instructions.

ARM Cortex-M3 ISA

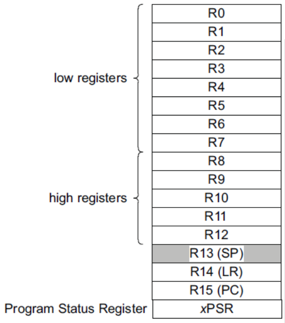
Instruction Set

`ADD Rd, Rn, <op2>`

Branching
Data processing
Load/Store
Exceptions
Miscellaneous



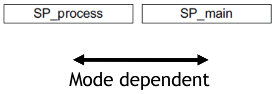
Registers



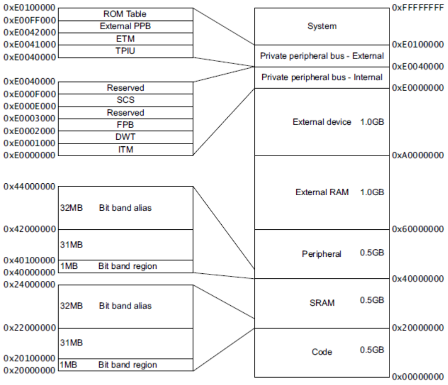
Note: there are two stack pointers!

SP_process (PSP) used by:
- Base app code (when not running an exception handler)

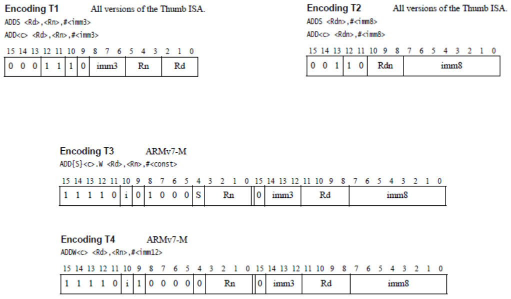
SP_main (MSP) used by:
- OS kernel
- Exception handlers
- App code w/ privileded access



Address Space



Instruction Encoding
ADD immediate



Encoding T1 All versions of the Thumb ISA.

```
ADDS <Rd>,<Rn>,#<imm3>
ADD<c> <Rd>,<Rn>,#<imm3>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

Encoding T2 All versions of the Thumb ISA.

```
ADDS <Rdn>,#<imm8>
ADD<c> <Rdn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

Encoding T3 ARMv7-M

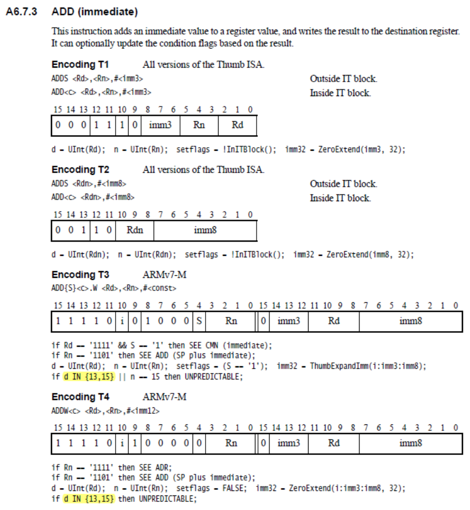
```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							



A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb ISA.

```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Branch

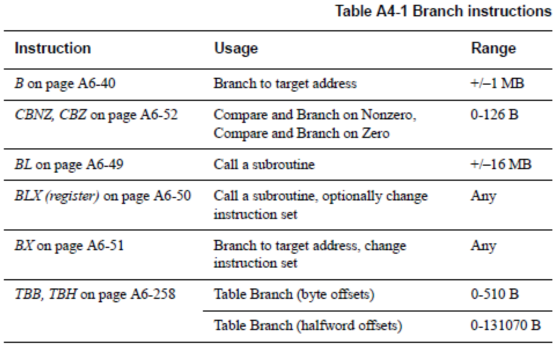


Table A4-1 Branch instructions

Instruction	Usage	Range
B on page A6-40	Branch to target address	+/–1 MB
CBNZ, CBZ on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
BL on page A6-49	Call a subroutine	+/–16 MB
BLX (register) on page A6-50	Call a subroutine, optionally change instruction set	Any
BX on page A6-51	Branch to target address, change instruction set	Any
TBB, TBH on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

Data processing instructions

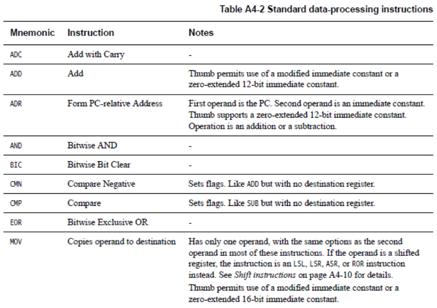


Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.

Many, Many More!

Load/Store instructions

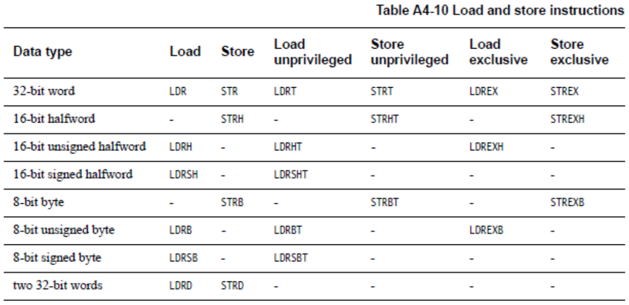


Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

Miscellaneous instructions

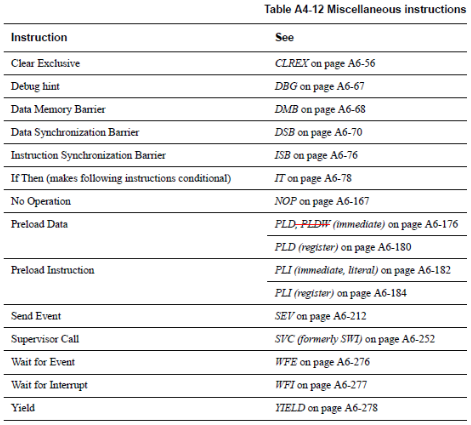


Table A4-12 Miscellaneous instructions

Instruction	See
Clear Exclusive	*CLREX* on page A6-56
Debug hint	*DBG* on page A6-67
Data Memory Barrier	*DMB* on page A6-68
Data Synchronization Barrier	*DSB* on page A6-70
Instruction Synchronization Barrier	*ISB* on page A6-76
If Then (makes following instructions conditional)	*IT* on page A6-78
No Operation	*NOP* on page A6-167
Preload Data	*PLD~~, PLDW~~ (immediate)* on page A6-176
	PLD (register) on page A6-180
Preload Instruction	*PLI (immediate, literal)* on page A6-182
	PLI (register) on page A6-184
Send Event	*SEV* on page A6-212
Supervisor Call	*SVC (formerly SWI)* on page A6-252
Wait for Event	*WFE* on page A6-276
Wait for Interrupt	*WFI* on page A6-277
Yield	*YIELD* on page A6-278

Addressing Modes (again)

- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>

<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>
- Lots of weird options…

A5.3.2 Modified immediate constants in Thumb instructions

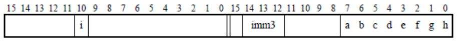


Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A5-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
. . .	. . 8-bit values shifted to other positions .
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

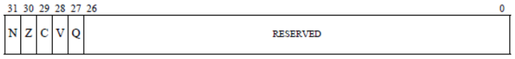
[a] In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

[b] UNPREDICTABLE if abcdefgh == 00000000.

ARMv7-M Architecture Reference Manual

ARMv7-M_ARM.pdf

Application Program Status Register (APSR)



APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- SUB Rx, Ry
  - Rx = Rx - Ry
  - APSR unchanged
- SUB<u>S</u>
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADD<u>S</u>
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated

Overflow and carry in APSR

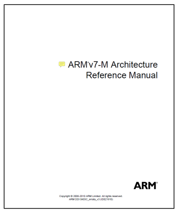
```
unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);

signed_sum = SInt(x) + SInt(y) + UInt(carry_in);

result = unsigned_sum<N-1:0>; // == signed_sum<N-1:0>

carry_out = if UInt(result) == unsigned_sum then ’0’ else ’1’;

overflow = if SInt(result) == signed_sum then ’0’ else ’1’;
```

Conditional execution:
Append to many instructions for conditional execution

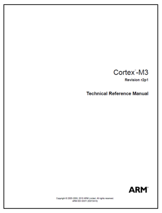
Table A6-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [ab]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [c]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [d]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [e]	Always (unconditional)	Always (unconditional)	Any

The ARM architecture “books” for this class

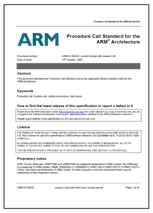

ARMv7-M Architecture Reference Manual

ARM


Cortex-M3

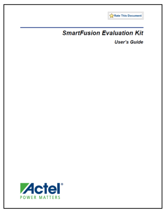
Technical Reference Manual

ARM

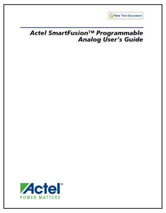

ARM Procedure Call Standard for the ARM® Architecture

Actel SmartFusion™ Microcontroller Subsystem User's Guide

Actel

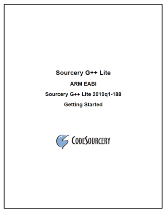

SmartFusion Evaluation Kit User's Guide

Actel


Actel SmartFusion™ Programmable Analog User's Guide

Actel

The ARM software tools “books” for this class

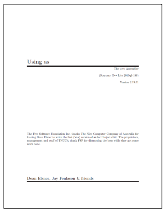

Sourcery G++ Lite

ARM EABI

Sourcery G++ Lite 2010q1-188

Getting Started

CodeSourcery


Using as

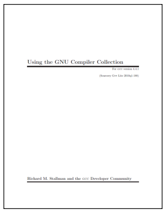
The GNU Assembler

(Sourcery G++ Lite 2010q1-188)

Version 2.19.51

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of as for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

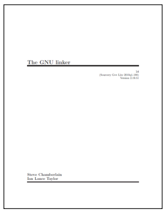
Dean Elsner, Jay Fenlason & friends


Using the GNU Compiler Collection

For GCC version 4.4.1

(Sourcery G++ Lite 2010q1-188)

Richard M. Stallman and the GCC Developer Community

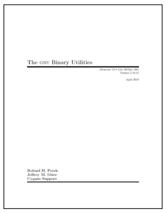

The GNU linker

ld

(Sourcery G++ Lite 2010q1-188)

Version 2.19.51

Steve Chamberlain
Ian Lance Taylor

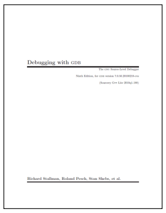

The GNU Binary Utilities

(Sourcery G++ Lite 2010q1-188)

Version 2.19.51

April 2010

Roland H. Pesch
Jeffrey M. Osier
Cygnus Support


Debugging with GDB

The GNU Source-Level Debugger

Ninth Edition, for GDB version 7.0.50.20100218-cvs

(Sourcery G++ Lite 2010q1-188)

Richard Stallman, Roland Pesch, Stan Shebs, et al.

Exercise:
What is the value of r2 at done?

```
...
start:
      movs r0, #1
      movs r1, #1
      movs r2, #1
      sub  r0, r1
      bne  done
      movs r2, #2
done:
      b    done
...
```

Solution:
what is the value of r2 at done?

```
...
start:
      movs r0, #1      // r0 ← 1, Z=0
      movs r1, #1      // r1 ← 1, Z=0
      movs r2, #1      // r2 ← 1, Z=0
      sub  r0, r1      // r0 ← r0-r1
                       // but Z flag untouched
                       // since sub vs subs
      bne  done        // NE true when Z==0
                       // So, take the branch
      movs r2, #2      // not executed
done:
      b    done        // r2 is still 1
...
```

Today…

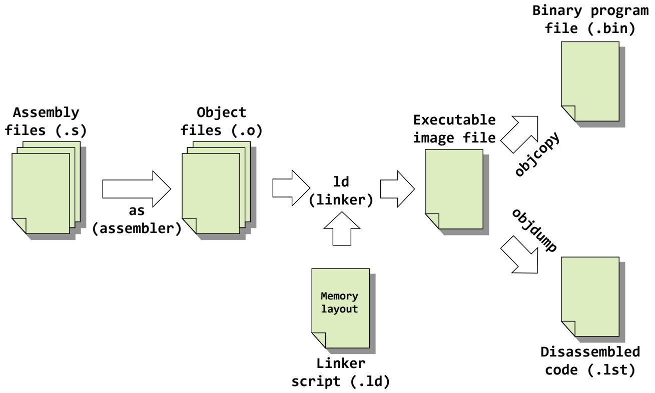
Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)

How does an assembly language program get turned into a executable program image?



What are the real GNU executable names for the ARM?

- Just add the prefix “arm-none-eabi-” prefix
- Assembler (as)
  - arm-none-eabi-as
- Linker (ld)
  - arm-none-eabi-ld
- Object copy (objcopy)
  - arm-none-eabi-objcopy
- Object dump (objdump)
  - arm-none-eabi-objdump
- C Compiler (gcc)
  - arm-none-eabi-gcc
- C++ Compiler (g++)
  - arm-none-eabi-g++

How are assembly files assembled?

- $ arm-none-eabi-as
  - Useful options
    - -mcpu
    - -mthumb
    - -o

```
$ arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

A “real” ARM assembly language program for GNU

```
        .equ    STACK_TOP, 0x20000800
        .text
        .syntax unified
        .thumb
        .global _start
        .type   start, %function

_start:
        .word   STACK_TOP, start
start:
        movs r0, #10
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

What’s it all mean?

```
        .equ    STACK_TOP, 0x20000800     /* Equates symbol to value */
        .text                             /* Tells AS to assemble region */
        .syntax unified                   /* Means language is ARM UAL */
        .thumb                            /* Means ARM ISA is Thumb */
        .global _start                    /* .global exposes symbol */
                                          /* _start label is the beginning */
                                          /* ...of the program region */
        .type   start, %function          /* Specifies start is a function */
                                          /* start label is reset handler */
_start:
        .word   STACK_TOP, start          /* Inserts word 0x20000800 */
                                          /* Inserts word (start) */
start:
        movs r0, #10                      /* We’ve seen the rest ... */
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

A simple (hardcoded) Makefile example

```
all:
        arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
        arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
        arm-none-eabi-objcopy -Obinary example1.out example1.bin
        arm-none-eabi-objdump -S example1.out > example1.lst
```

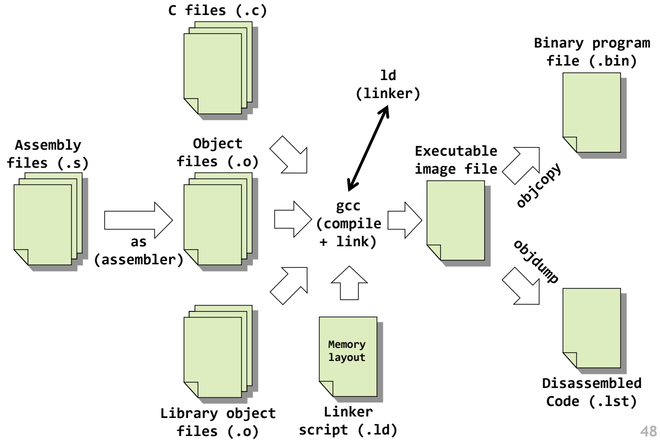
What information does the disassembled file provide?

```
all:
        arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
        arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
        arm-none-eabi-objcopy -Obinary example1.out example1.bin
        arm-none-eabi-objdump -S example1.out > example1.lst
```

```
          .equ      STACK_TOP, 0x20000800
          .text
          .syntax   unified
          .thumb
          .global   _start
          .type     start, %function

_start:
          .word     STACK_TOP, start
start:
          movs r0, #10
          movs r1, #0
loop:
          adds r1, r0
          subs r0, #1
          bne  loop
deadloop:
          b    deadloop
          .end
```

```
example1.out:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:       20000800   .word      0x20000800
   4:       00000009   .word      0x00000009

00000008 <start>:
   8:       200a       movs       r0, #10
   a:       2100       movs       r1, #0

0000000c <loop>:
   c:       1809       adds       r1, r1, r0
   e:       3801       subs       r0, #1
  10:       d1fc       bne.n      c <loop>

00000012 <deadloop>:
  12:       e7fe       b.n        12 <deadloop>
```

How does a mixed C/Assembly program get turned into a executable program image?



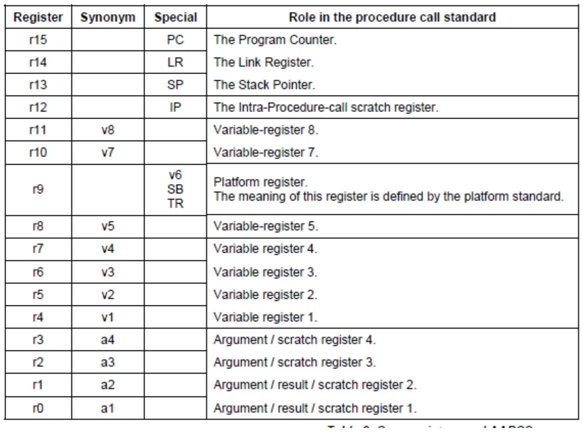
Today…

Finish ARM assembly example from last time

Walk though of the ARM ISA

Software Development Tool Flow

Application Binary Interface (ABI)



Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ABI quote

- A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP (and r9 in PCS variants that designate r9 as v6).

Questions?

Comments?

Discussion?