EECS 373
Design of Microprocessor-Based Systems

Branden Ghena
University of Michigan

Lecture 3: Assembly, Tools, and ABI
September 9, 2014

Slides developed in part by
Mark Brehob & Prabal Dutta

Announcements

- I’m not Prabal
  - You probably noticed
- Homework 1 is due
- No office hours this week
- Projects
  - Continue thinking about them

Today…

Finish ARM assembly example from last time

Software Development Tool Flow

Application Binary Interface (ABI)

Exercise:
What is the value of r2 at done?

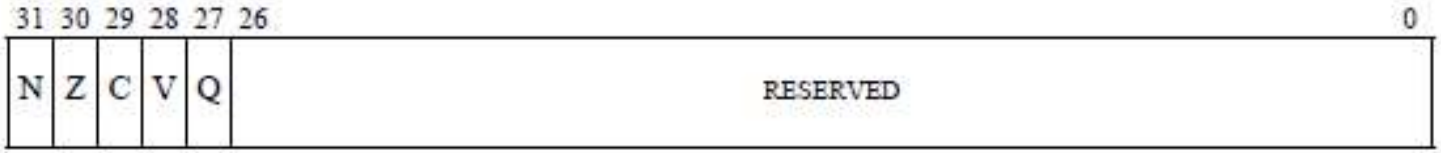
```
...
start:
      movs r0, #1
      movs r1, #1
      movs r2, #1
      sub  r0, r1
      bne  done
      movs r2, #2
done:
      b    done
...
```

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

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.

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
...
```

Real assembly example

```
        .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    /* Sets symbol to value (#define)*/
        .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
```

What happens after a power-on-reset (POR)?

- ARM Cortex-M3 (many others are similar)

- Reset procedure
  - SP ← mem(0x00000000)
  - PC ← mem(0x00000004)

```
_start:
    .word __STACKTOP            /* Top of Stack */
    .word Reset_Handler         /* Reset Handler */
    .word NMI_Handler           /* NMI Handler */
    .word HardFault_Handler     /* Hard Fault Handler */
    .word MemManage_Handler     /* MPU Fault Handler */
    .word BusFault_handler      /* Bus Fault Handler */
    ...
```

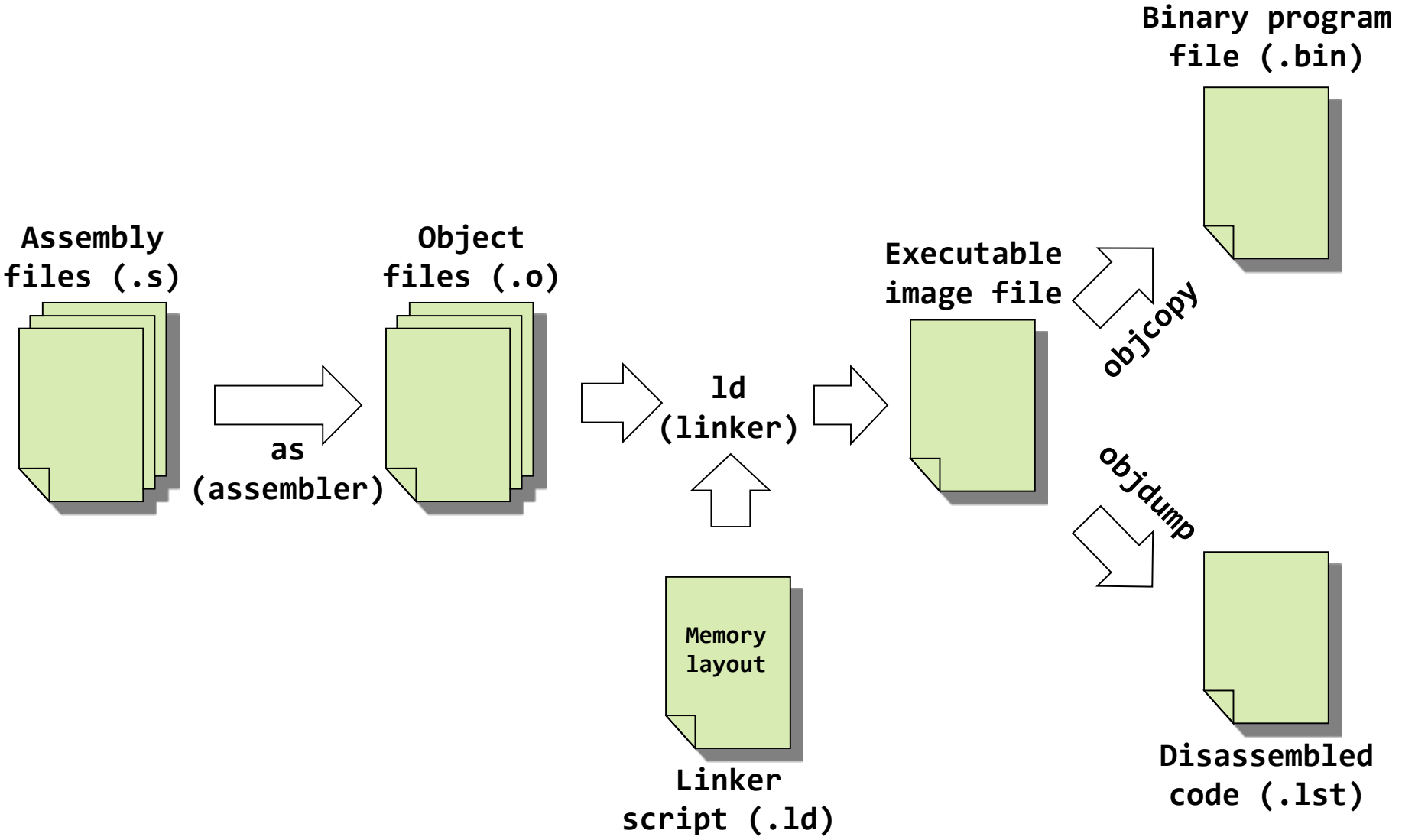
Today…

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++

Real-world example

- To the terminal!

(code at https://github.com/brghena/eecs373_toolchain_examples)

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 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>
```

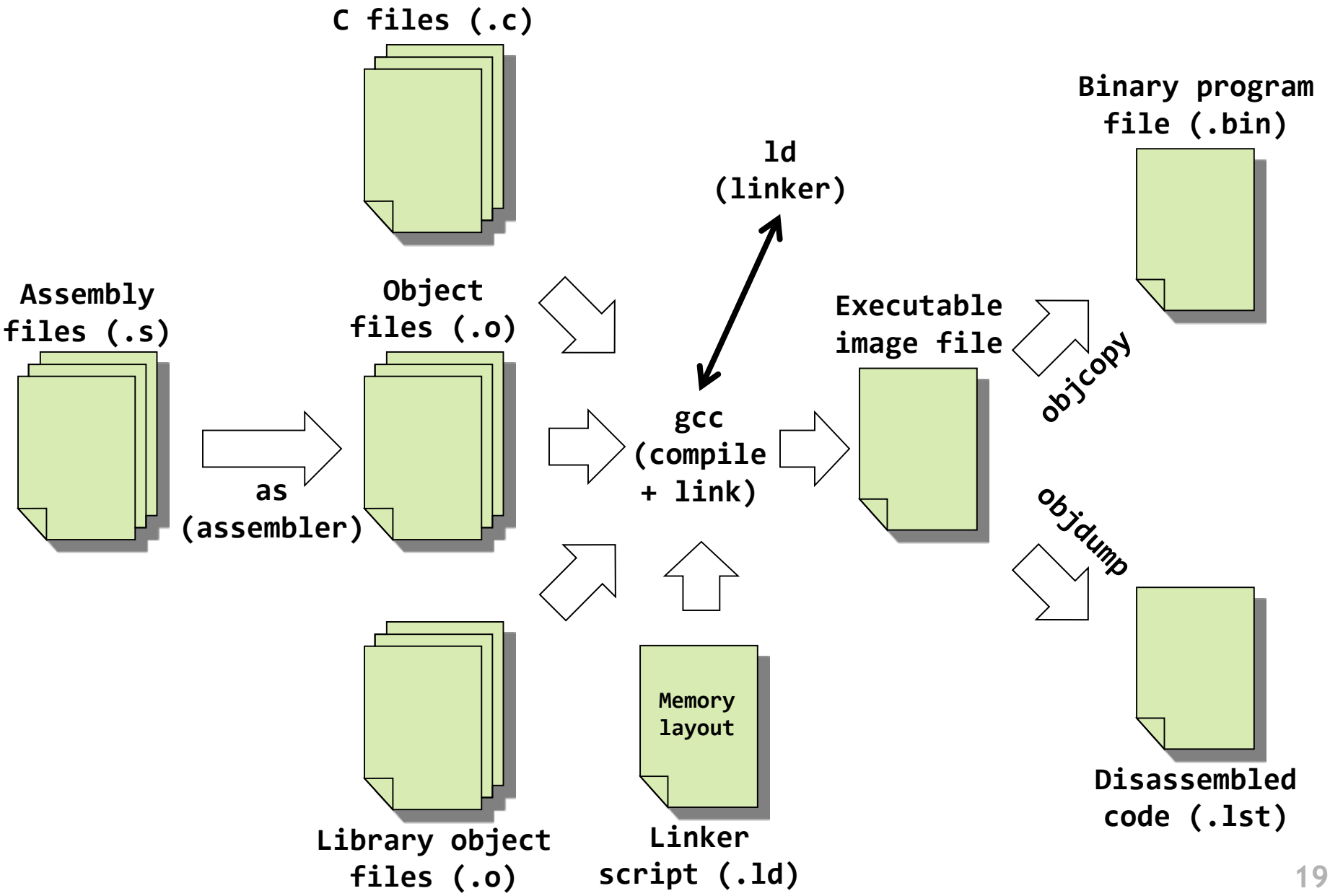
Linker script

```
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
 /* SmartFusion internal eSRAM */
 ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
 .text :
 {
    . = ALIGN(4);
     *(.text*)
    . = ALIGN(4);
     _etext = .;
 } >ram
}
end = .;
```

- Specifies little-endian arm in ELF format.
- Specifies ARM CPU
- Should start executing at label named “main”
- We have 64k of memory starting at 0x20000000. You can read, write and execute out of it. We’ve named it “ram”

- “.” is a reference to the current memory location
- First align to a word (4 byte) boundary
- Place all sections that include .text at the start (* here is a wildcard)
- Define a label named _etext to be the current address.
- Put it all in the memory location defined by the ram memory location.

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



Real-world example #2

- To the terminal! Again!

(code at https://github.com/brghena/eecs373_toolchain_examples)

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 Basic Rules

1. A subroutine must preserve the contents of the registers r4-11 and SP
   - Let's be careful with r9 though.

2. Arguments are passed though r0 to r3
   - If we need more, we put a pointer into memory in one of the registers.
     - We'll worry about that later.

3. Return value is placed in r0
   - r0 and r1 if 64-bits.

4. Allocate space on stack as needed. Use it as needed. Put it back when done…
   - Keep word aligned.

Let's write a simple ABI routine

- int bob(int a, int b)
  - returns $a^2 + b^2$
- Instructions you might need
  - add adds two values
  - mul multiplies two values
  - bx branch to register

Other useful factoids

- Stack grows down.
  - And pointed to by "sp"
- Address we need to go back to in "lr"

Register	Synonym
r15	
r14	
r13	
r12	
r11	v8
r10	v7
r9	
r8	v5
r7	v4
r6	v3
r5	v2
r4	v1
r3	a4
r2	a3
r1	a2
r0	a1

When is this relevant?

- The ABI is a contract with the compiler
  - All assembled C code will follow this standard
- You need to follow it if you want C and Assembly to work together correctly
- What if you are writing everything in Assembly by hand?
  - Maybe less important. Unless you're ever going to extend the code

Questions?

Comments?

Discussion?