Assembly / Compilation Review

Andrew Lukefahr
Indiana University - Bloomington

Does a CPU execute C Code?

not directly

→ translation into assembly first

→ translated into machine code

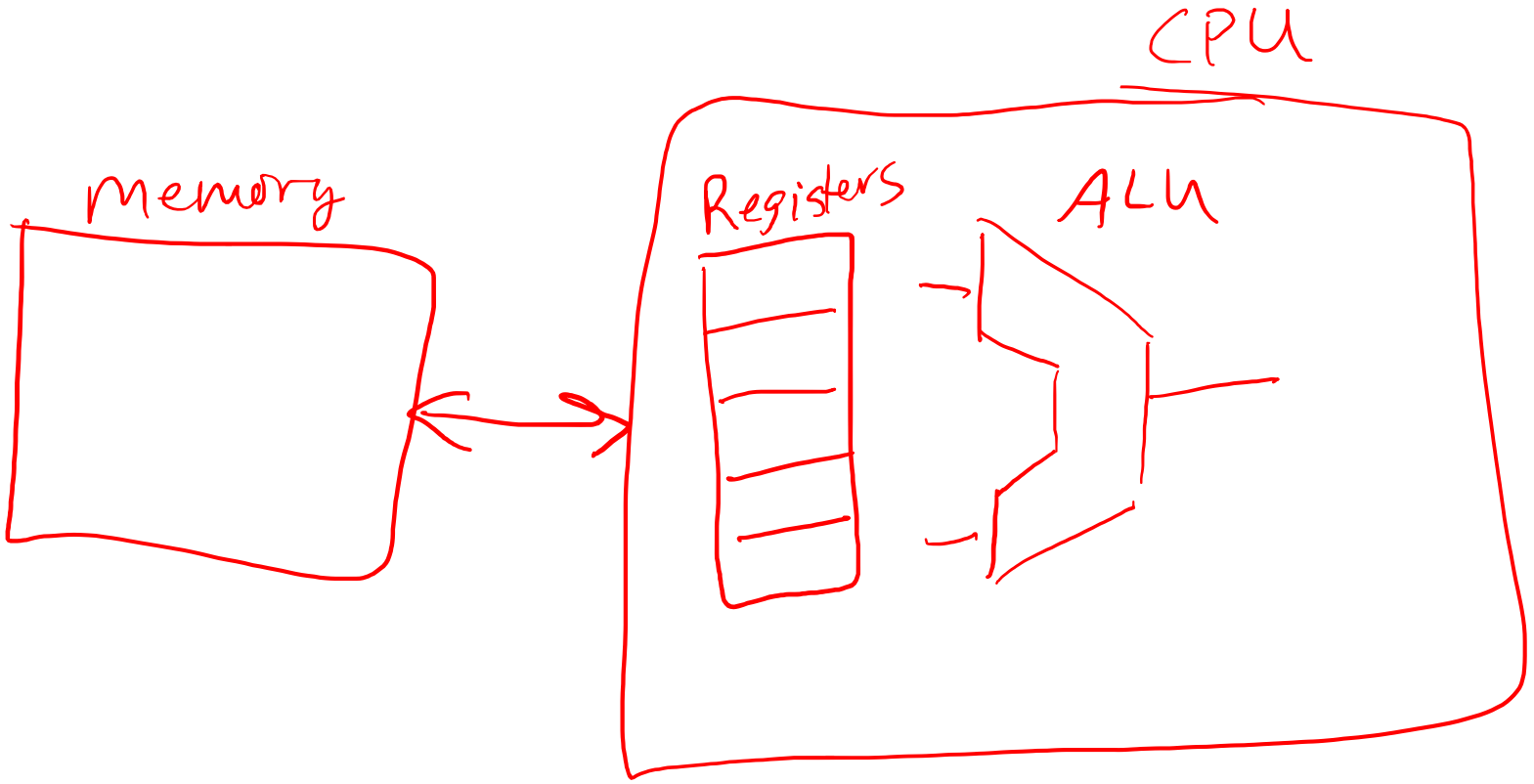
Instruction Set Architecture (ISA)

- What is it?
- What ISAs have you seen?

Engr : x86
CS : ARM

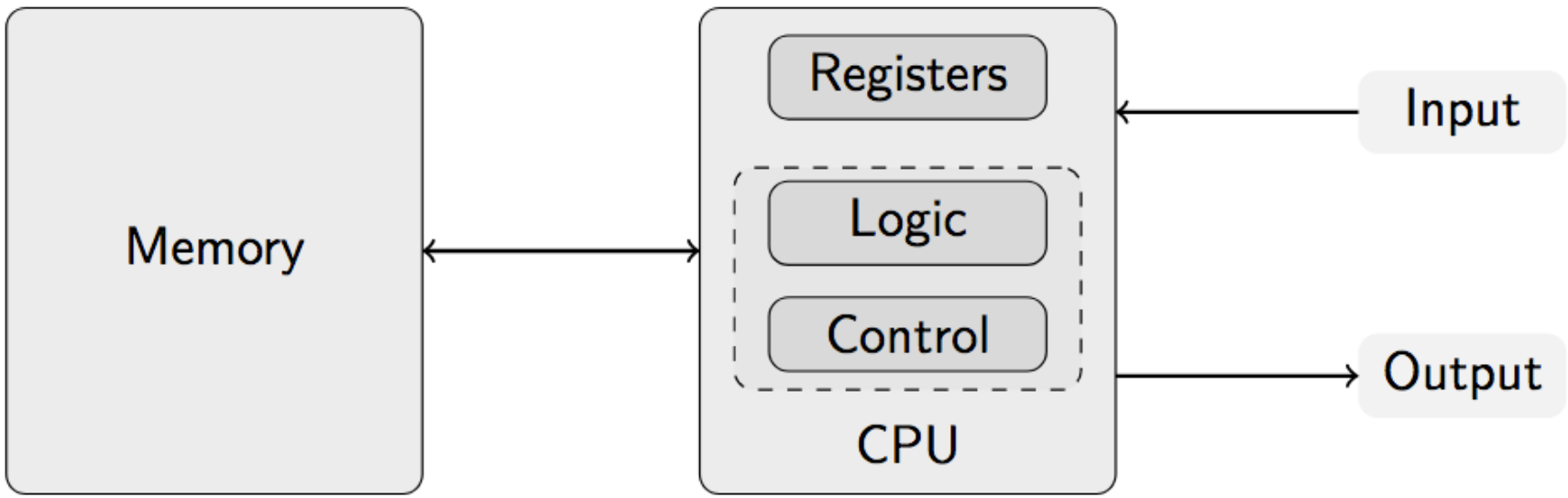
- We're going to do ARM (ARMv6m-Thumb2) in this class
  - Much simpler than x86

What is in a “Processor” ?



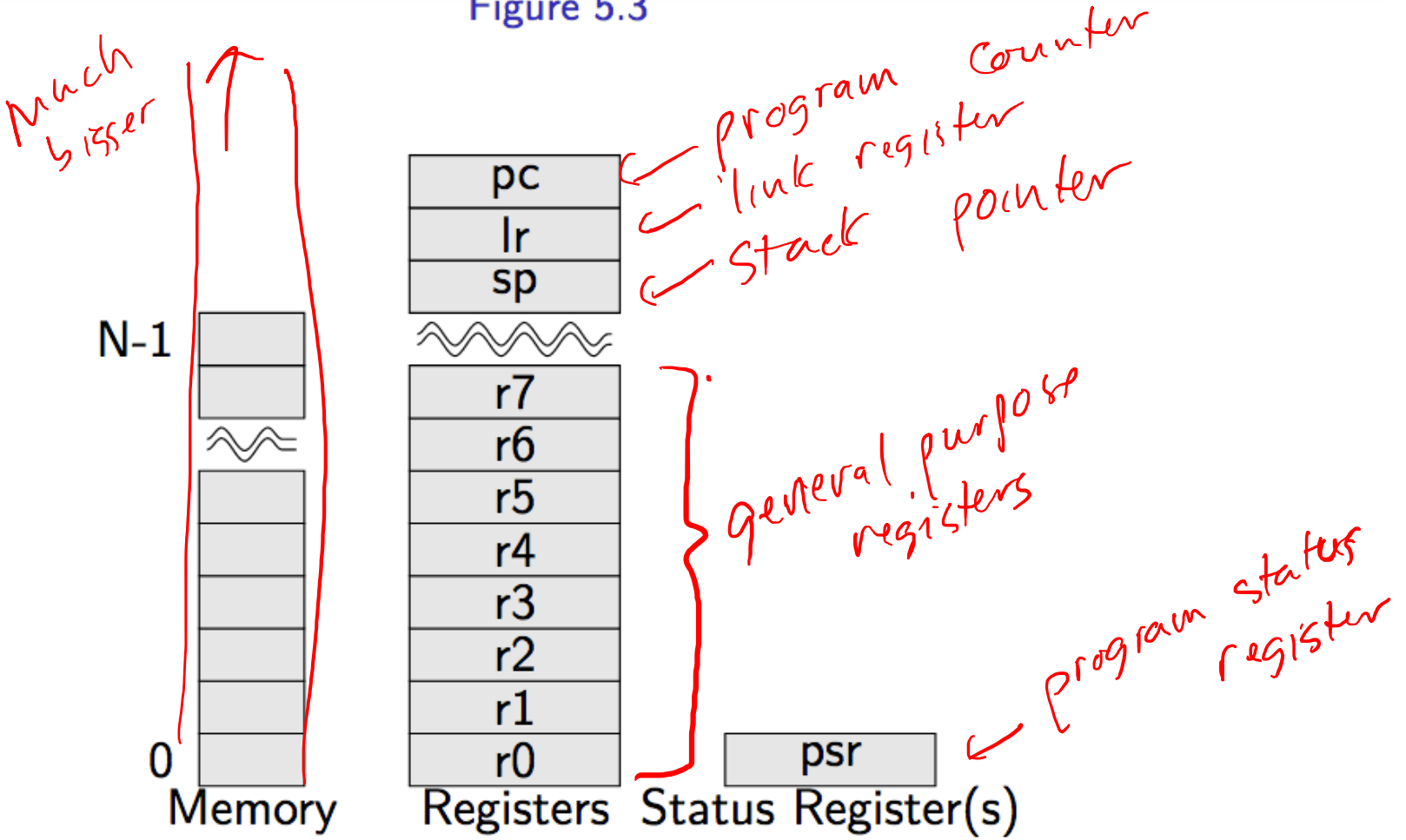
Stored Program (von Neumann) Machine

Figure 5.1

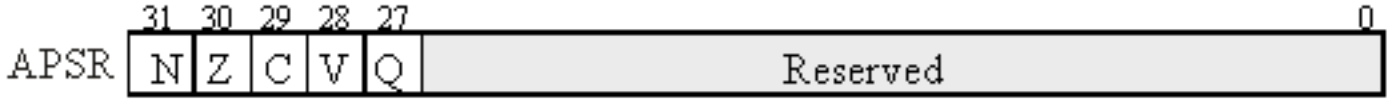


Cortex-M0 Programmer Model (Simplified)

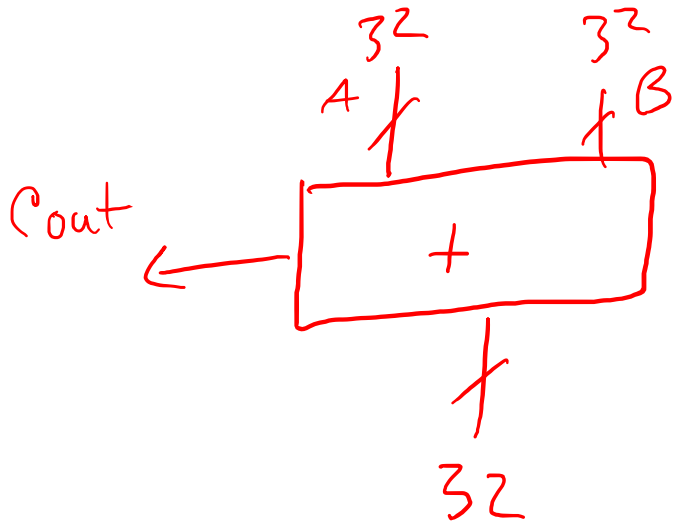
Figure 5.3



(Application) Program Status Register



- N: Negative
- Z: Zero
- C: Carry
- V: Overflow
- Q: Saturation (ignore)



Application Program Status Register

- N: Negative
  - Set to bit[31] of the ALU result
- Z: Zero
  - Set to 1 if ALU result == 0x0
- C: Carry
  - Carry out bit of ALU result
- V: Overflow
  - Set to 1 if signed ALU overflow occurred (cout ^ c[31])

Condition Flags

- **N** Negative – this bit is set when the result of an operation is negative (i.e. bit 31 is 1).
- **Z** Zero – this bit is set when the result of an operation is zero.
- **C** Carry – this bit is set when an (arithmetic) operation has a carry out.
- **V** oVerflow – this bit is set when the result of an arithmetic operation has the wrong sign and therefore does not fit in 32 bits. For example, the addition of two positive numbers yielding a negative result. Only occurs when adding numbers with the same sign or subtracting numbers with opposite sign.

The compare operation in the previous example is implemented using subtraction (`r0 - 0`) and discarding the result, but keeping the side effect – setting the four condition flags.

Instruction Interpretation

Figure 5.2

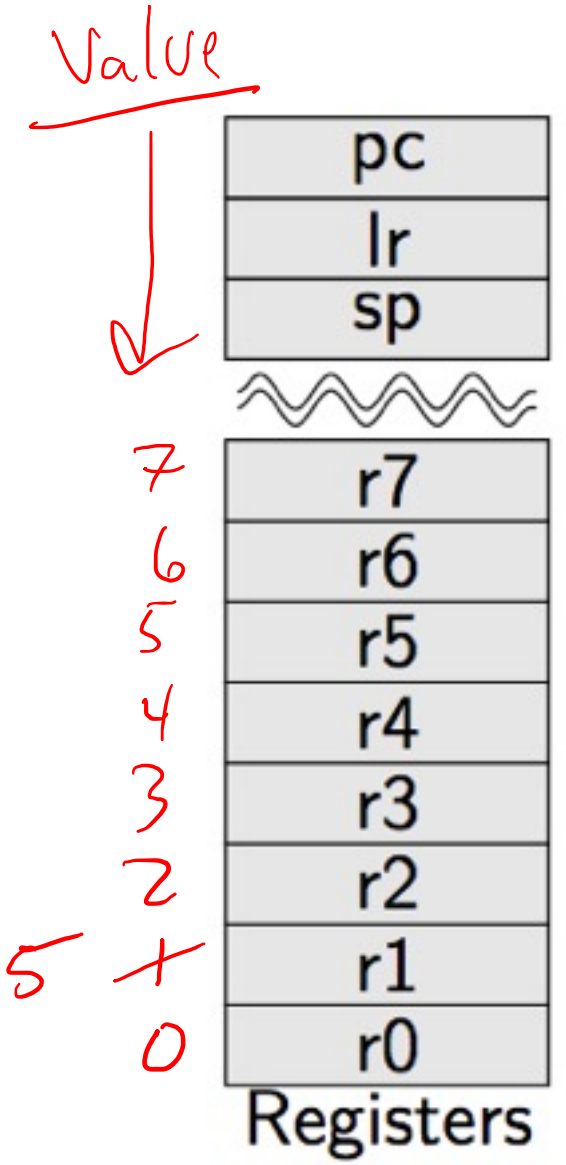
```
extern inst_t M[];          // the memory
extern unsigned int pc;    // index into the memory

while (1) {
      inst_t inst;    // an instruction
      inst = M[pc++];
      interpret(inst);
}
```

Operations for registers?

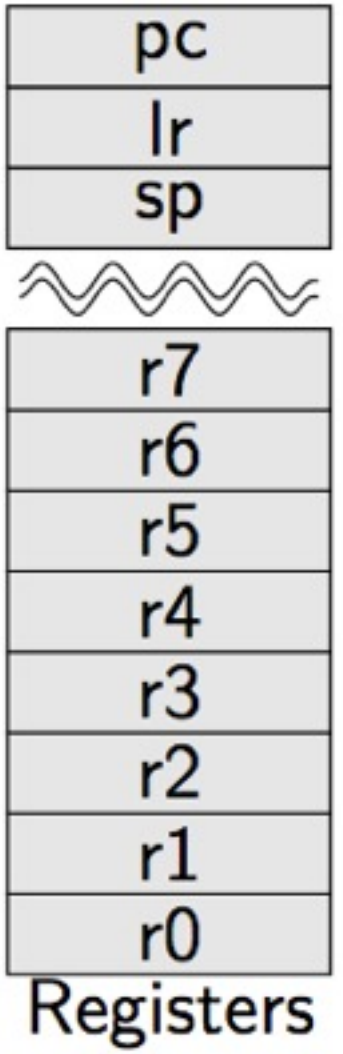
C-like ⇒ r1 = r2 + r3

Assembly → adds r1, r2, r3



C -> Assembly

- C:
  - `z = x + y;`
- Assembly:
  - `adds r0, r1, r2`



Addition Instructions

adds Rd, Rn, imm3	Rd = Rn + imm3
adds Rd, imm8	Rd = Rd + imm8
adds Rd, Rn	Rd = Rd + Rn
add Rd, Rn	Rd = Rd + Rn [1]
adds Rd, Rn, Rm	Rd = Rn + Rm
add sp, imm7	sp = sp + (imm7«2)
add Rn, sp, imm8	Rn = sp + (imm8«2)
adcs Rd, Rm	Rd = Rd + Rm + Carry

[1] One of Rd, Rn must be a high register – r8-r15.

Assembly Instructions

- There a ~100 assembly instructions

ARMv6M - Thumb2

- We'll cover some more of them later
  - But not all of them

- Logic
- Arithmetic
- Branch

C -> Assembly

- C:

```
32 int multiply( int x, int y)
33 {
34     int z = 0;
35     for (int i = 0; i < x; ++i){
36         z = z + y;
37     }
38     return z;
39 }
```

4 3

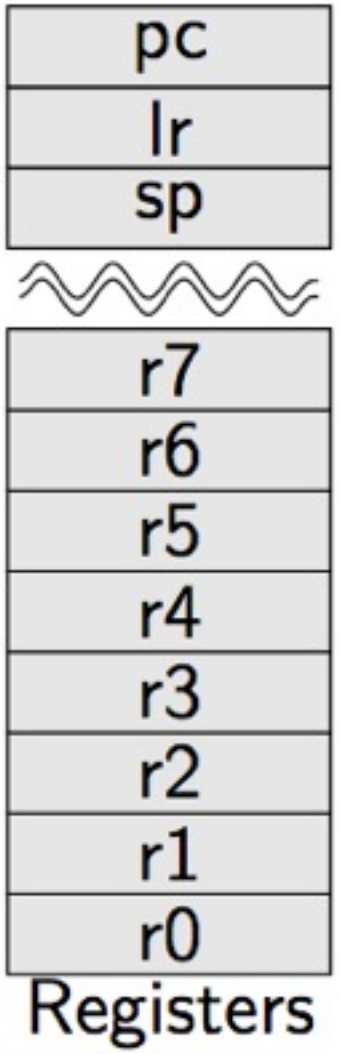
- Assembly:

```
cmp r0, #0
ble.n   5c <multiply+0x10>
movs    r3, #0
adds    r3, #1
cmp r0, r3
bne.n   52 <multiply+0x6>
muls    r0, r1
b.n 5e <multiply+0x12>
```

Whoops....

Smart Compiler...

r0 = r0 * r1



GCC TOOL FLOW

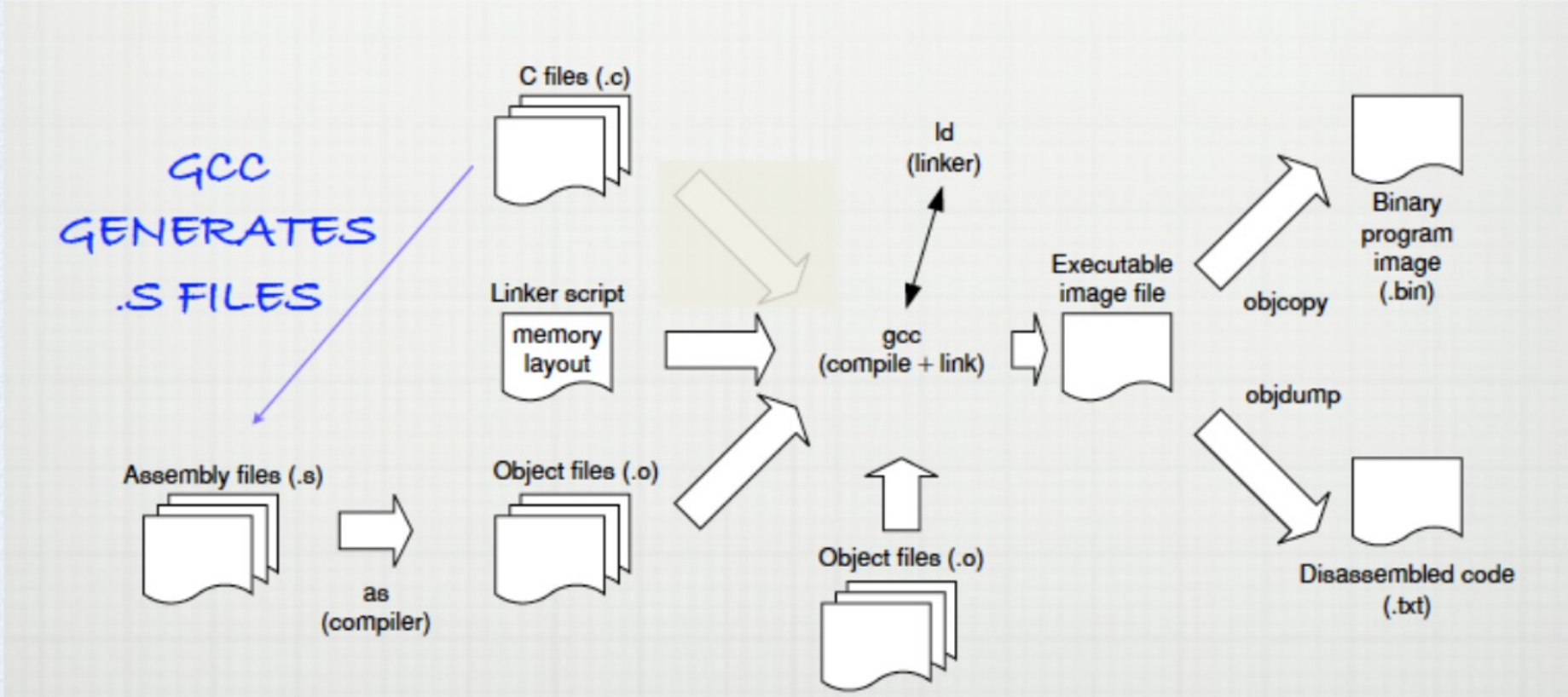


FIGURE 19.1

Example Development Flow Based on the CodeSourcery G++ Tool Chain.

Figure modified from "The Definitive Guide to the ARM CORTEX-M3" ©Elsevier

GCC STAGES

- **Compilation**: each **source** (`.c`) file is compiled into an **assembly** (`.s`) file.
- **Assembly**: each assembly file is assembled into a *relocatable* **object** (`.o`) file.
- **Linking**: program and library object files are combined into a single **executable** file (`a.out` by default).
  - Some *dynamically linked* library files are not incorporated with the executable code until the latter is loaded, or even until they are first used.
- By default, `gcc` generates executable and deletes assembly and object files when it is done with them.
- With `-c` option, `gcc` saves object files and does not generate executable unless told to with `-o` option.

AS

- Create processor specific object files from assembly language.
- Examples:

Assembly Language Input File

```
$ cat t1.s
    .text
    .syntax unified
    .thumb
    .global  inc
    .type    inc, %function
inc:
    adds r0,r0,#1
    bx   lr
$
```

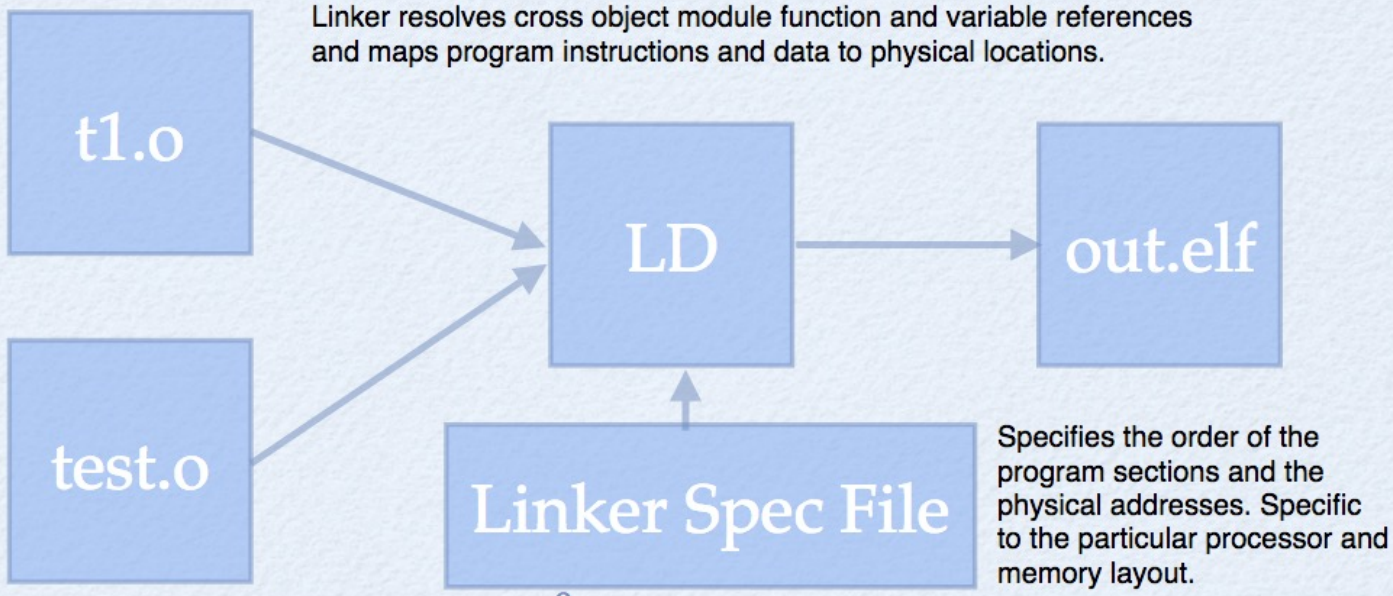
Assembler Command with Options

```
$ arm-none-eabi-as -g -mcpu=cortex-m0 -mthumb t1.s -o t1.o
```

Options:
-g: include debugging output
-mcpu=cortex-m0: generate Cortex M0 instructions
-mthumb: Generate thumb code
t1.s: Assembly language file to use as input
t1.o: Object file output

LD

- Combine object files a single executable, resolves program references, and creates an executable program with a memory layout specified by a linker specification file.



LINKER SPECIFICATION

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x2000A000;    /* end of 40K RAM */

/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200;      /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x8000000, LENGTH = 256K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 40K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
}

/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */

     <Continued Output…>
```

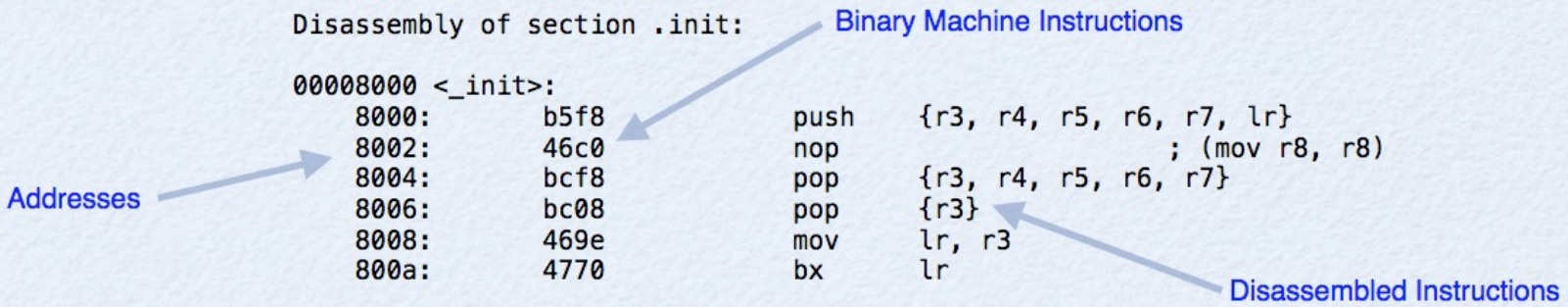
OBJDUMP

- Displays information about an object file. Useful to disassemble an object file or elf executable.
- Examples:

```
$ arm-none-eabi-objdump -d t1.o
t1.o:     file format elf32-littlearm
Disassembly of section .text:
00000000 <inc>:
   0:3001       adds  r0, #1
   2:4770       bx lr

$ arm-none-eabi-objdump -d base_example.elf > disassembled_file.txt
```

OBJDUMP OUTPUT




```
base_example.elf:     file format elf32-littlearm


Disassembly of section .init:

00008000 <_init>:
    8000:       b5f8            push    {r3, r4, r5, r6, r7, lr}
    8002:       46c0            nop                     ; (mov r8, r8)
    8004:       bcf8            pop     {r3, r4, r5, r6, r7}
    8006:       bc08            pop     {r3}
    8008:       469e            mov     lr, r3
    800a:       4770            bx      lr

Disassembly of section .text:

00008010 <exit>:
    8010:       b510            push    {r4, lr}
    8012:       2100            movs    r1, #0
    8014:       1c04            adds    r4, r0, #0
    8016:       f000 f929       bl      826c <__call_exitprocs>
    801a:       4b04            ldr     r3, [pc, #16]   ; (802c <exit+0x1c>)
    801c:       6818            ldr     r0, [r3, #0]
    801e:       6bc3            ldr     r3, [r0, #60]   ; 0x3c

   <continued …>
```

C -> Assembly -> Object Code

```
32 int multiply( int x, int y)
33 {
34     int z = 0;
35     for (int i = 0; i < x; ++i){
36         z = z + y;
37     }
38     return z;
39 }
```

```
cmp     r0, #0
ble.n   10 <multiply+0x10>
movs    r3, #0
adds    r3, #1
cmp     r0, r3
bne.n   6 <multiply+0x6>
muls    r0, r1
b.n     12 <multiply+0x12>
movs    r0, #0
bx      lr
```

```
00000000 <multiply>:
   0:   2800
   2:   dd05
   4:   2300
   6:   3301
   8:   4298
   a:   d1fc
   c:   4348
   e:   e000
  10:   2000
  12:   4770
```

Object Code -> Binary

```
00000000 <multiply>:
   0:   2800
   2:   dd05
   4:   2300
   6:   3301
   8:   4298
   a:   d1fc
   c:   4348
   e:   e000
  10:   2000
  12:   4770
```

```
20 0000004c: dd052800
21 00000050: 33012300
22 00000054: d1fc4298
23 00000058: e0004348
24 0000005c: 47702000
```

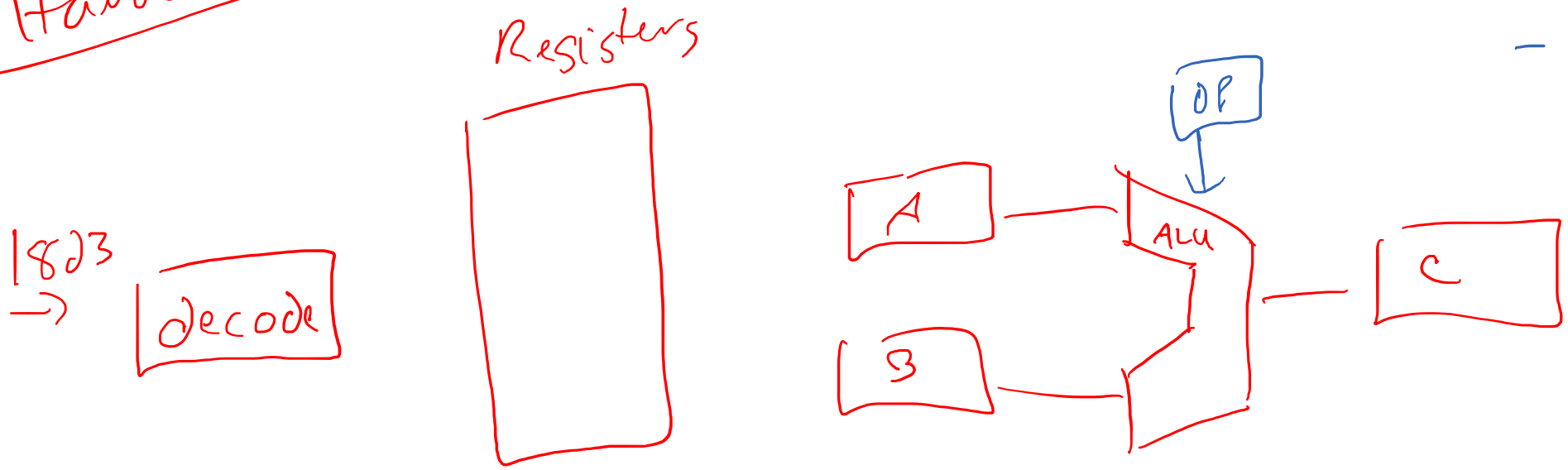
Demo Time

Software

0x18d3

adds r2, r3, r3

Hardware



Decoder

- what is A, B
- what is OP?
- what does C go?

Next Time

- C -> Assembly -> Object -> Binary
- How do CPUs execute Binaries?