EECS 373

Design of Microprocessor-Based Systems

Ron Dreslinski
University of Michigan

Lecture 3: Toolchain, ABI, Memory Mapped I/O
September 14th, 2016

Slides developed in part by
Prof. Dutta and Dr. Brehob

Announcements

- HW2 Released on Friday, Due 9/26
- Exam and Project Meeting Rooms/Times
  - Project Meeting: Monday October 10, 6:30-8pm, 1003 EECS
  - Midterm Exam: Monday October 24, 7-9pm, 1500 EECS
- **Want to learn more about hands-on embedded systems?**
  - https://www.eecs.umich.edu/hub/
  - Lessons on Soldering, PCB design, and using simple boards (Arduino, etc.)
  - Office hours staffed by PhD students in embedded systems.
    - Useful for talking to about projects (either class or own).

Outline

- Assembly wrap-up
- Tool chain
- ABI
- Basic memory-mapped I/O

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
1	1	1	1	0	i	0	1	0	0	0	S	Rn			

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

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

Application Program Status Register (APSR)

31	30	29	28	27	26 … 0
N	Z	C	V	Q	RESERVED

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
- SUBS
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated

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®

ARM DDI 0403C_errata_v3 (ID021910)

Cortex™-M3
Revision r2p1

Technical Reference Manual

ARM®

ARM DDI 0337I (ID072410)

ARM

Procedure Call Standard for the
ARM® Architecture

Document number: ARM IHI 0042D, current through ABI release 2.08
Date of Issue: 16th October, 2009

Abstract

This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the ARM architecture.

Keywords

Procedure call function call, calling conventions, data layout

How to find the latest release of this specification or report a defect in it

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than one year old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection).

Please report defects in this specification to *arm* dot *eabi* at *arm* dot *com*.

Licence

THE TERMS OF YOUR ROYALTY FREE LIMITED LICENCE TO USE THIS ABI SPECIFICATION ARE GIVEN IN SECTION **1.4**, ***Your licence to use this specification*** (ARM contract reference **LEC-ELA-00081 V2.0**). PLEASE READ THEM CAREFULLY.

BY DOWNLOADING OR OTHERWISE USING THIS SPECIFICATION, YOU AGREE TO BE BOUND BY ALL OF ITS TERMS. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

THIS ABI SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES (SEE SECTION 1.4 FOR DETAILS).

Proprietary notice

ARM, Thumb, RealView, ARM7TDMI and ARM9TDMI are registered trademarks of ARM Limited. The ARM logo is a trademark of ARM Limited. ARM9, ARM926EJ-S, ARM946E-S, ARM1136J-S ARM1156T2F-S ARM1176JZ-S Cortex, and Neon are trademarks of ARM Limited. All other products or services mentioned herein may be trademarks of their respective owners.

ARM IHI 0042D Page 1 of 34

Rate This Document

Actel SmartFusion™ Microcontroller Subsystem User’s Guide

Actel® POWER MATTERS

Rate This Document

SmartFusion Evaluation Kit
User’s Guide

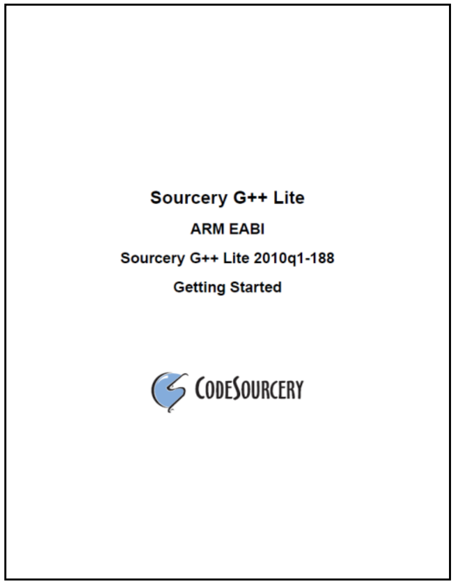
Actel® POWER MATTERS

Rate This Document

Actel SmartFusion™ Programmable Analog User’s Guide

Actel® POWER MATTERS

The ARM software tools "books" for this class



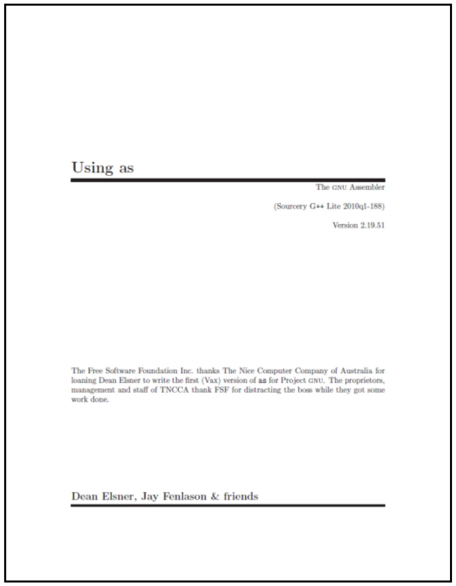
Sourcery G++ Lite

ARM EABI

Sourcery G++ Lite 2010q1-188

Getting Started

CodeSourcery



Using as

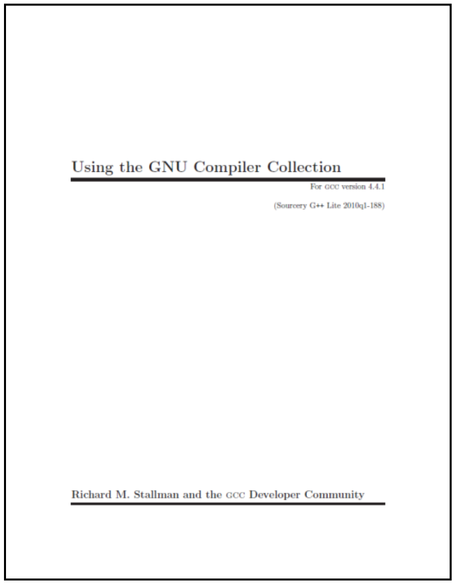
The GNU Assembler

(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

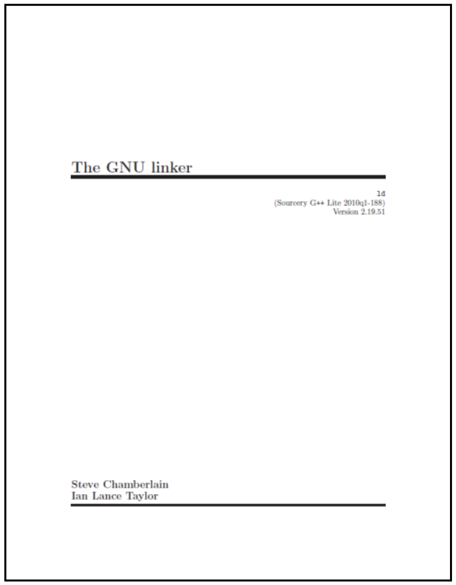


Using the GNU Compiler Collection

For GCC version 4.4.1

(Sourcery G++ Lite 2010q1-188)

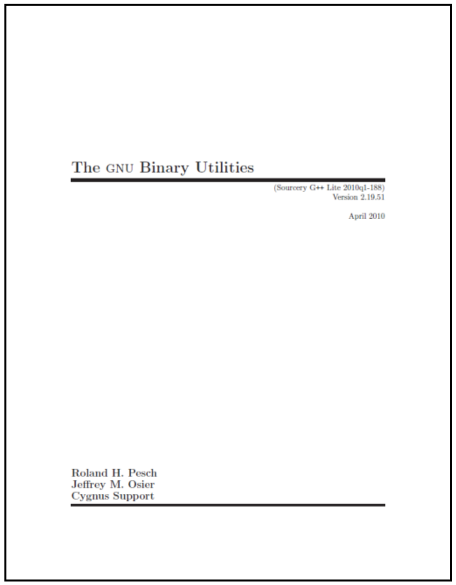
Richard M. Stallman and the GCC Developer Community



The GNU linker

ld
(Sourcery G++ Lite 2010q1-188)
Version 2.19.51

Steve Chamberlain
Ian Lance Taylor

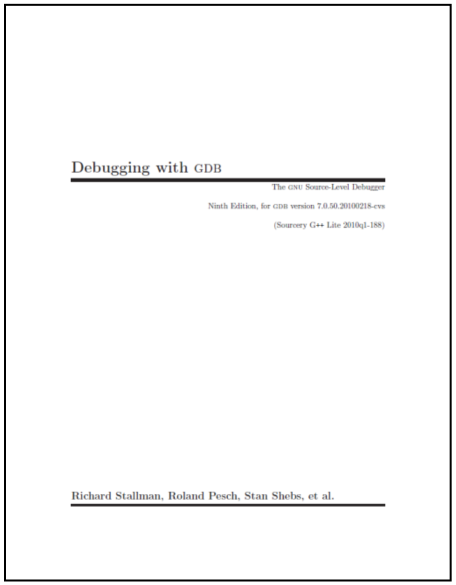


The GNU Binary Utilities

(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

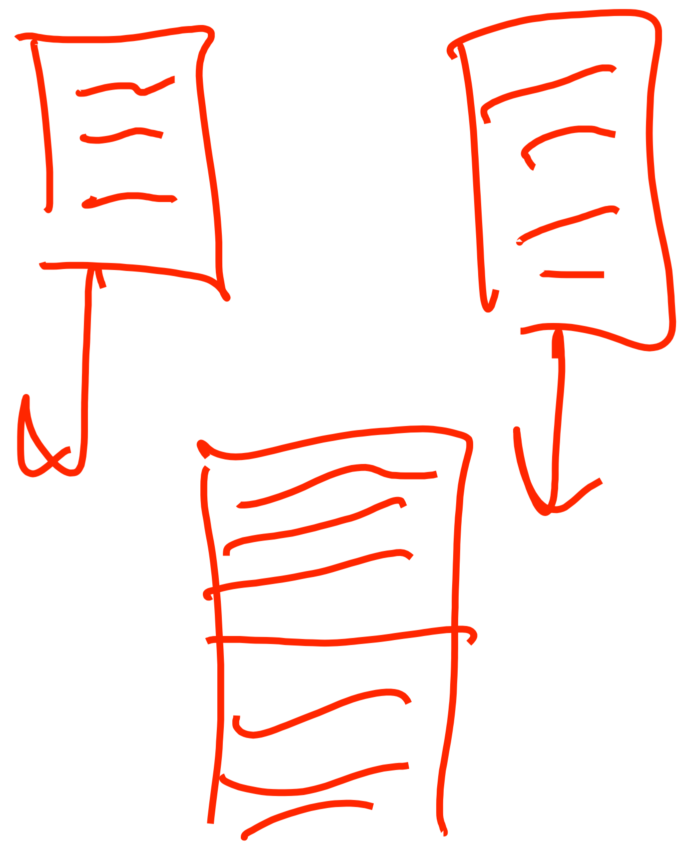
Ninth Edition, for GDB version 7.0.50.20100218-cvs

(Sourcery G++ Lite 2010q1-188)

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

An 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
```



Disassembled object code

```
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>
```

0009

0008

0a 20

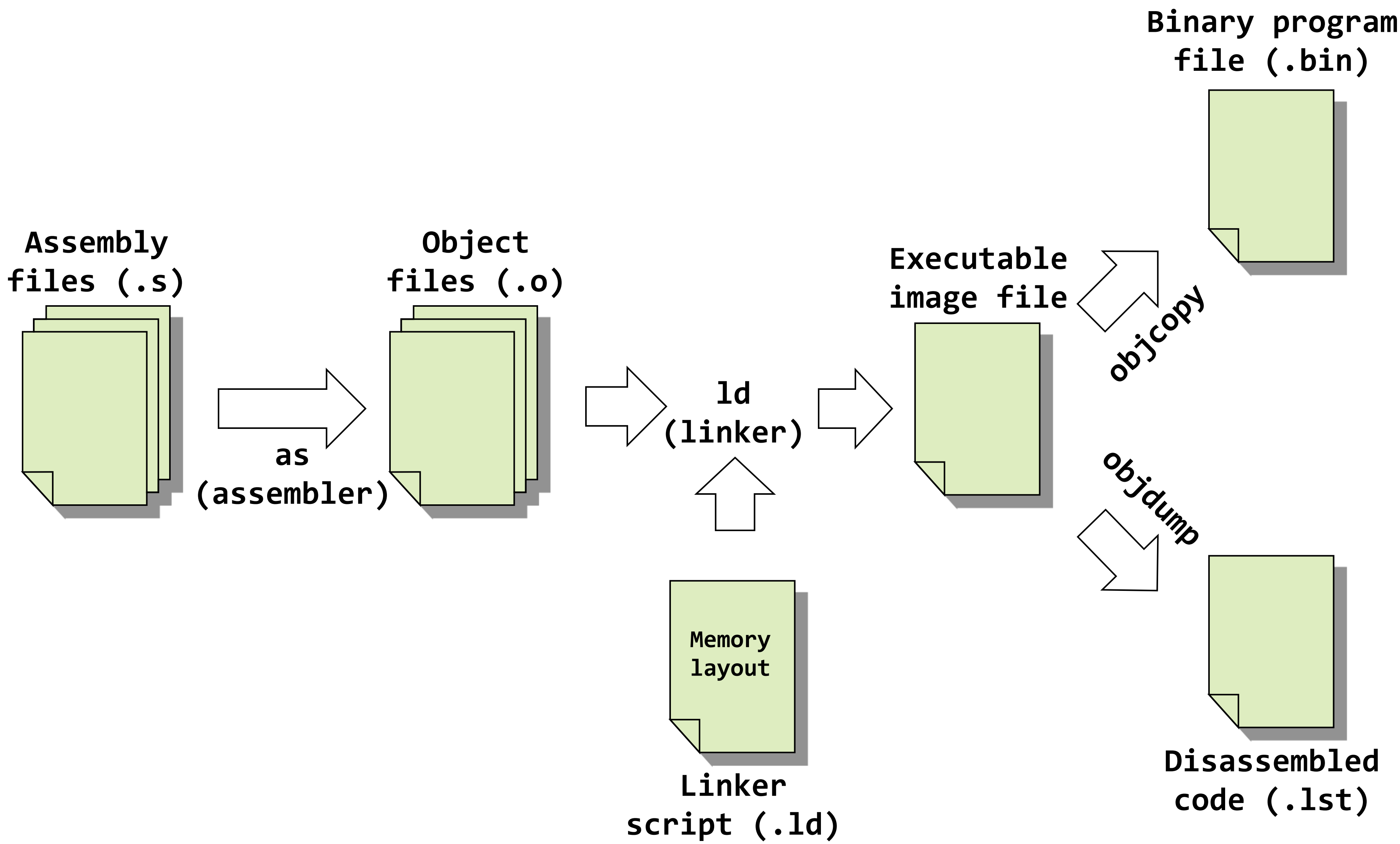
Outline

- Review
- Tool chain
- ABI
- Basic memory-mapped I/O

A simple Makefile

```
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 example.bin
	arm-none-eabi-objdump -S example1.out > example1.list
```

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



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) boundry
- 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.

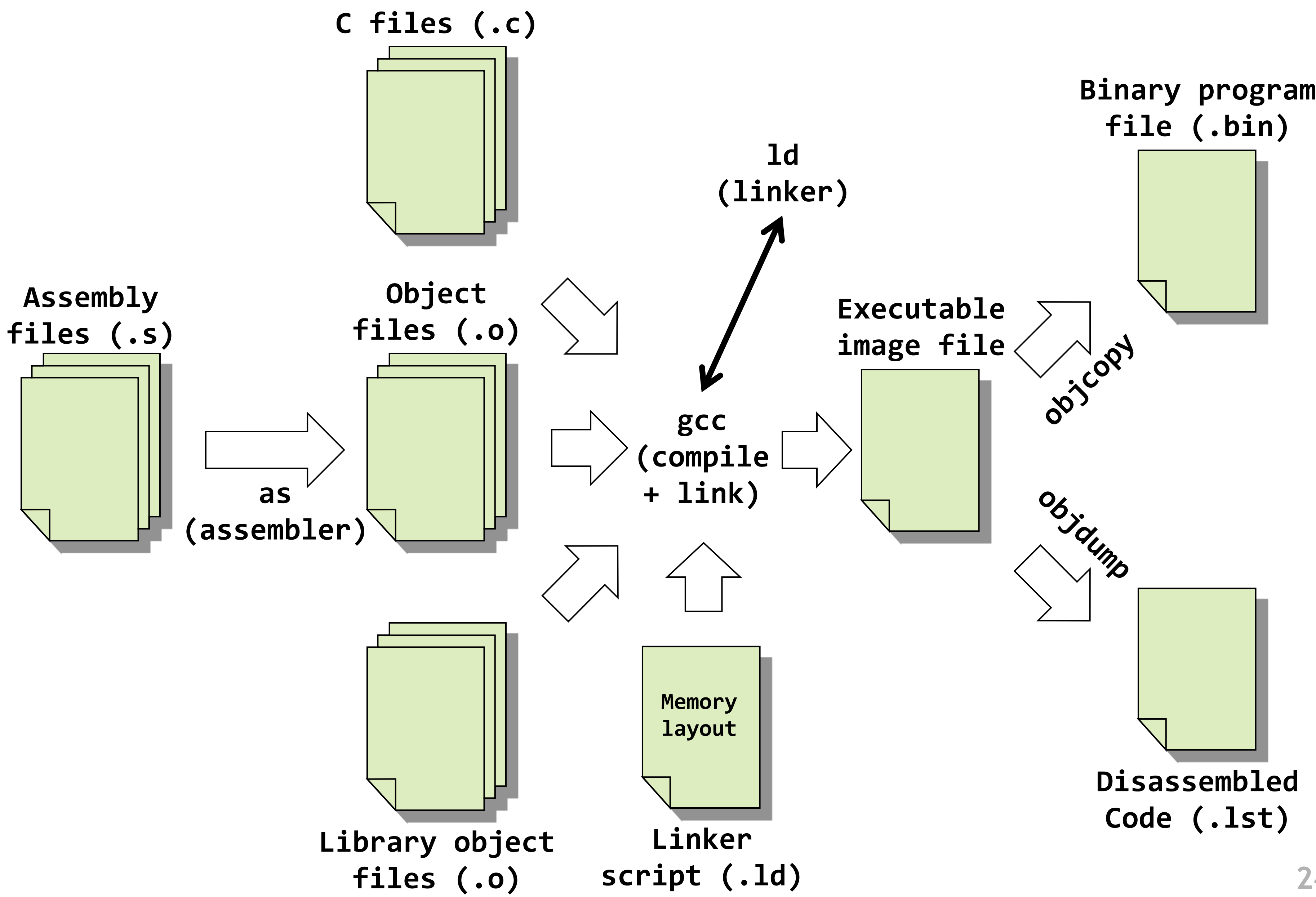
What information does the disassembled file provide?

```
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?



Outline

- Review
- Tool chain
- ABI
- Basic memory-mapped I/O

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.

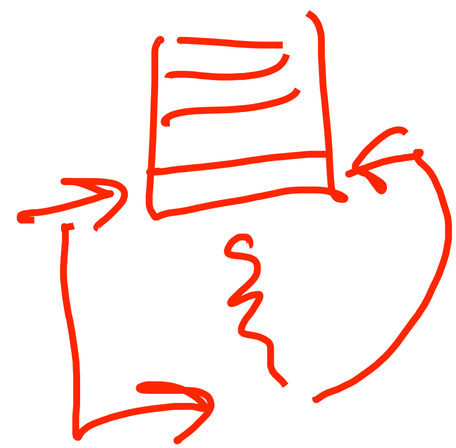
v6

r0

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.

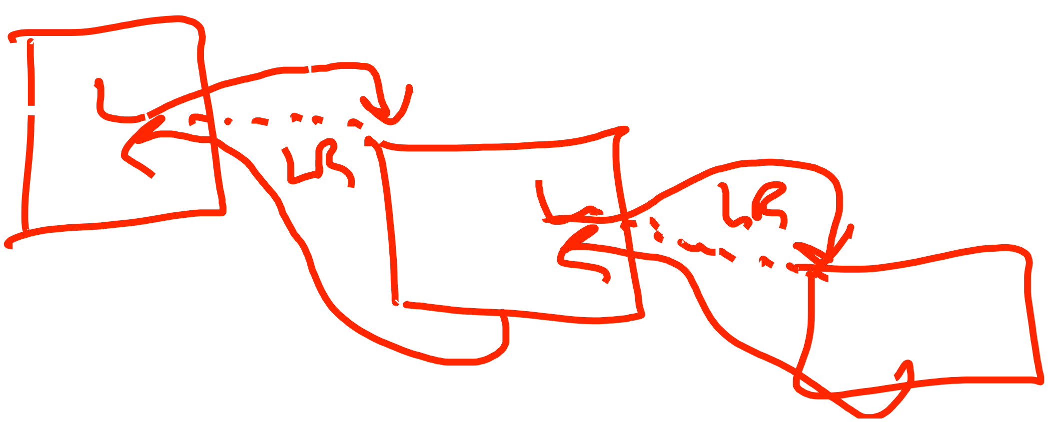
Other useful factoids

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

And useful things for the example

- Assembly instructions
  - add adds two values
  - mul multiplies two values
  - bx branch to register

bne label



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

reg a in r0
reg b in r1
reg result in r0

```
bob: mul r0, r0, r0
     mul r1, r1, r1
     add r0, r1, r0
     bx  lr (r14)
```

Same thing, but for no good reason using the stack

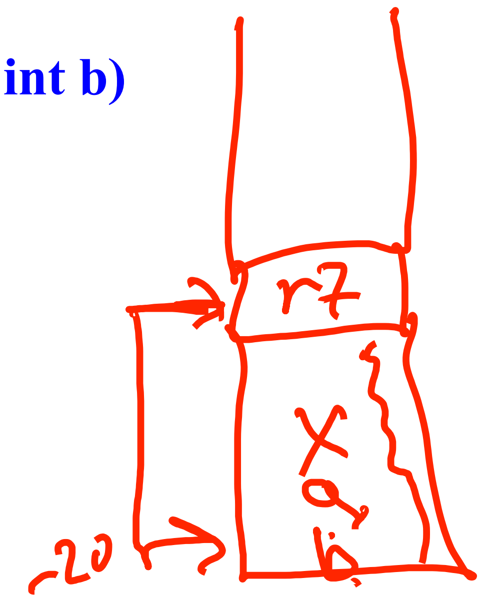
- int bob(int a, int b)
  - returns $a^2 + b^2$

Some disassembly

- 0x20000490 <bob>: push {r7}
- 0x20000492 <bob+2>: sub sp, #20
- 0x20000494 <bob+4>: add r7, sp, #0
- 0x20000496 <bob+6>: str r0, [r7, #4]
- 0x20000498 <bob+8>: str r1, [r7, #0]
- x=a*a;
- 0x2000049a <bob+10>: ldr r3, [r7, #4]
- 0x2000049c <bob+12>: ldr r2, [r7, #4]
- 0x2000049e <bob+14>: mul.w r3, r2, r3
- 0x200004a2 <bob+18>: str r3, [r7, #8]
- y=b*b;
- 0x200004a4 <bob+20>: ldr r3, [r7, #0]
- 0x200004a6 <bob+22>: ldr r2, [r7, #0]
- 0x200004a8 <bob+24>: mul.w r3, r2, r3
- 0x
- x=x+y;
- 0x200004ae <bob+30>: ldr r2, [r7, #8]
- 0x200004b0 <bob+32>: ldr r3, [r7, #12]
- 0x200004b2 <bob+34>: add r3, r2
- 0x200004b4 <bob+36>: str r3, [r7, #8]
- 0x200004ac <bob+28>: str r3, [r7, #12]

- return(x);
- 0x200004b6 <bob+38>: ldr r3, [r7, #8]
- }
- 0x200004b8 <bob+40>: mov r0, r3
- 0x200004ba <bob+42>: add.w r7, r7, #20
- 0x200004be <bob+46>: mov sp, r7
- 0x200004c0 <bob+48>: pop {r7}
- 0x200004c2 <bob+50>: bx lr

```
int bob(int a, int b)
{
    int x, y;
    x=a*a;
    y=b*b;
    x=x+y;
    return(x);
}
```



Outline

- Review
- Tool chain
- ABI
- Basic memory-mapped I/O

Hello

Memory-mapped I/O

- The idea is really simple
  - Instead of real memory at a given memory address, have an I/O device respond.
    - Huh?
- Example:
  - Let's say we want to have an LED turn on if we write a "1" to memory location 5.
  - Further, let's have a button we can read (pushed or unpushed) by reading address 4.
    - If pushed, it returns a 1.
    - If not pushed, it returns a 0.

Now…

- How do you get that to happen?
  - We could just say “magic” but that’s not very helpful.
  - Let’s start by detailing a simple bus and hooking hardware up to it.
- We’ll work on a real bus next time!

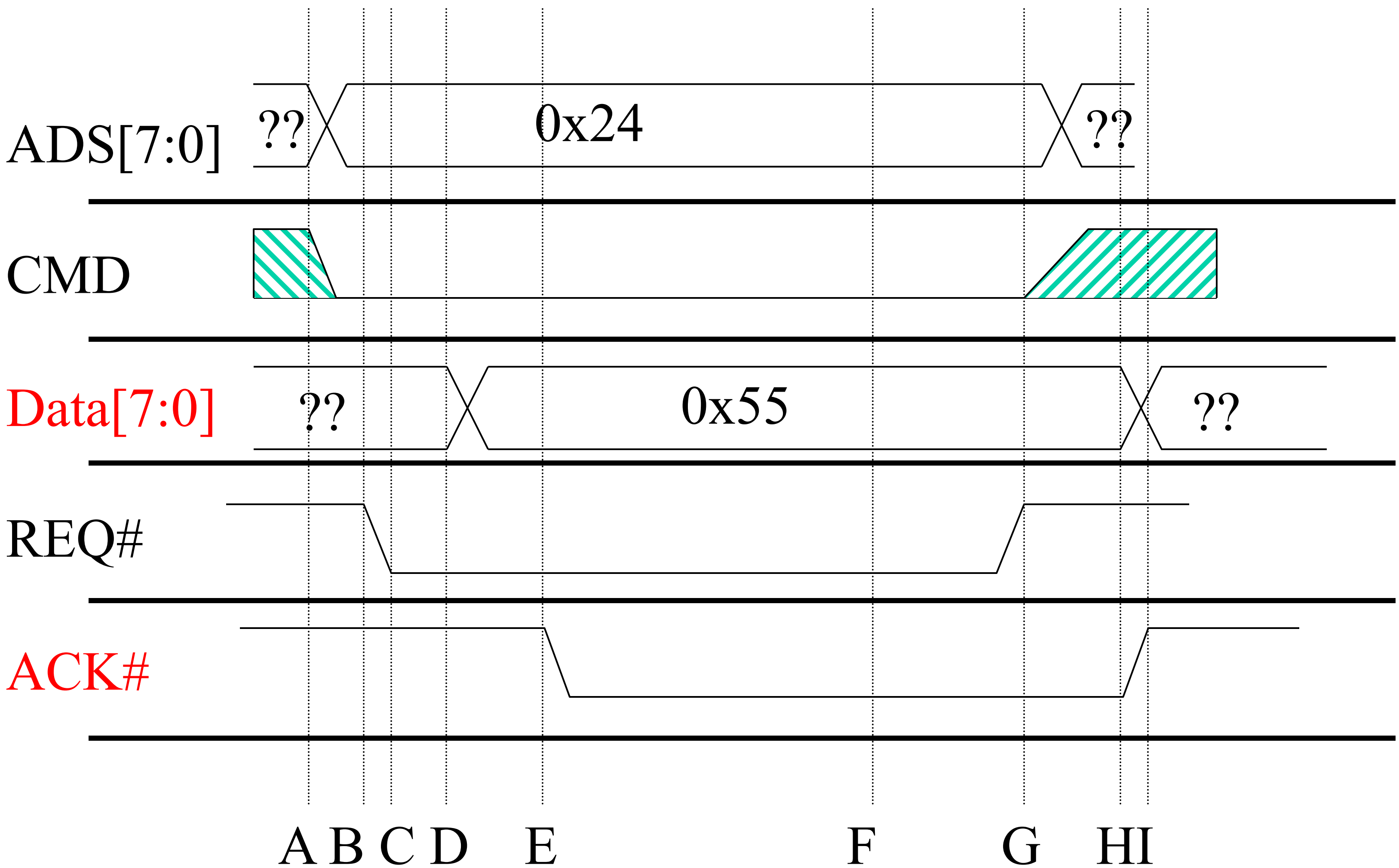
Basic example

- Discuss a basic bus protocol
  - Asynchronous (no clock)
  - Initiator and Target
  - REQ#, ACK#, Data[7:0], ADS[7:0], CMD
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is requesting something.
    - ACK# low means target has done its job.

A read transaction

- Say initiator wants to read location 0x24
  - Initiator sets ADS=0x24, CMD=0.
  - Initiator *then* sets REQ# to low. (why do we need a delay? How much of a delay?)
  - Target sees read request.
  - Target drives data onto data bus.
  - Target *then* sets ACK# to low.
  - Initiator grabs the data from the data bus.
  - Initiator sets REQ# to high, stops driving ADS and CMD
  - Target stops driving data, sets ACK# to high terminating the transaction

Read transaction



A write transaction
(write 0xF4 to location 0x31)

– Initiator sets ADS=0x31, CMD=1, Data=0xF4
– Initiator *then* sets REQ# to low.
– Target sees write request.
– Target reads data from data bus. (Just has to store in a register, need not write all the way to memory!)
– Target *then* sets ACK# to low.
– Initiator sets REQ# to high & stops driving other lines.
– Target sets ACK# to high terminating the transaction

The push-button

(if ADS=0x04 write 0 or 1 depending on button)

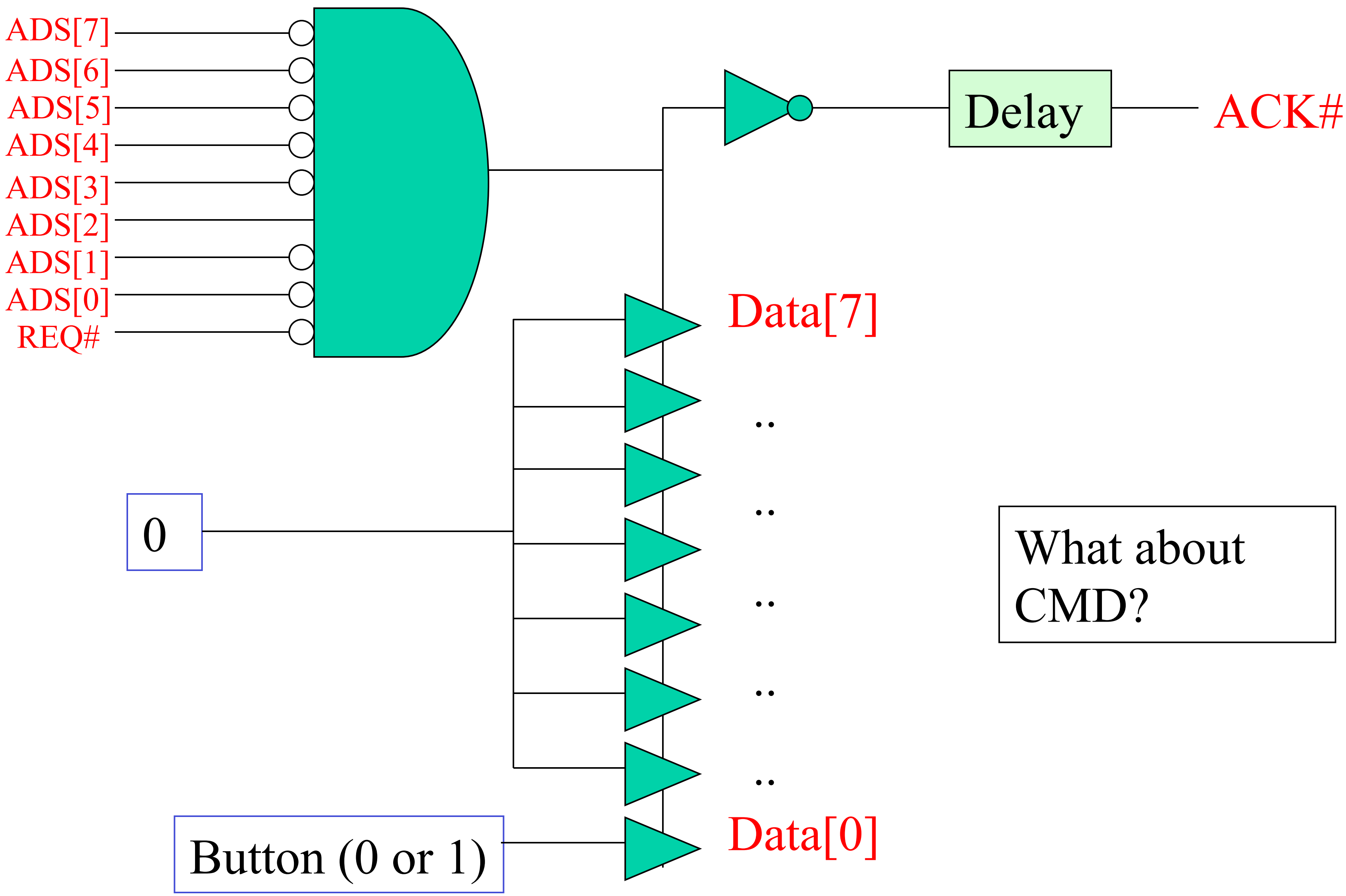
ADS[7]
ADS[6]
ADS[5]
ADS[4]
ADS[3]
ADS[2]
ADS[1]
ADS[0]
REQ#

ACK#

Button (0 or 1)

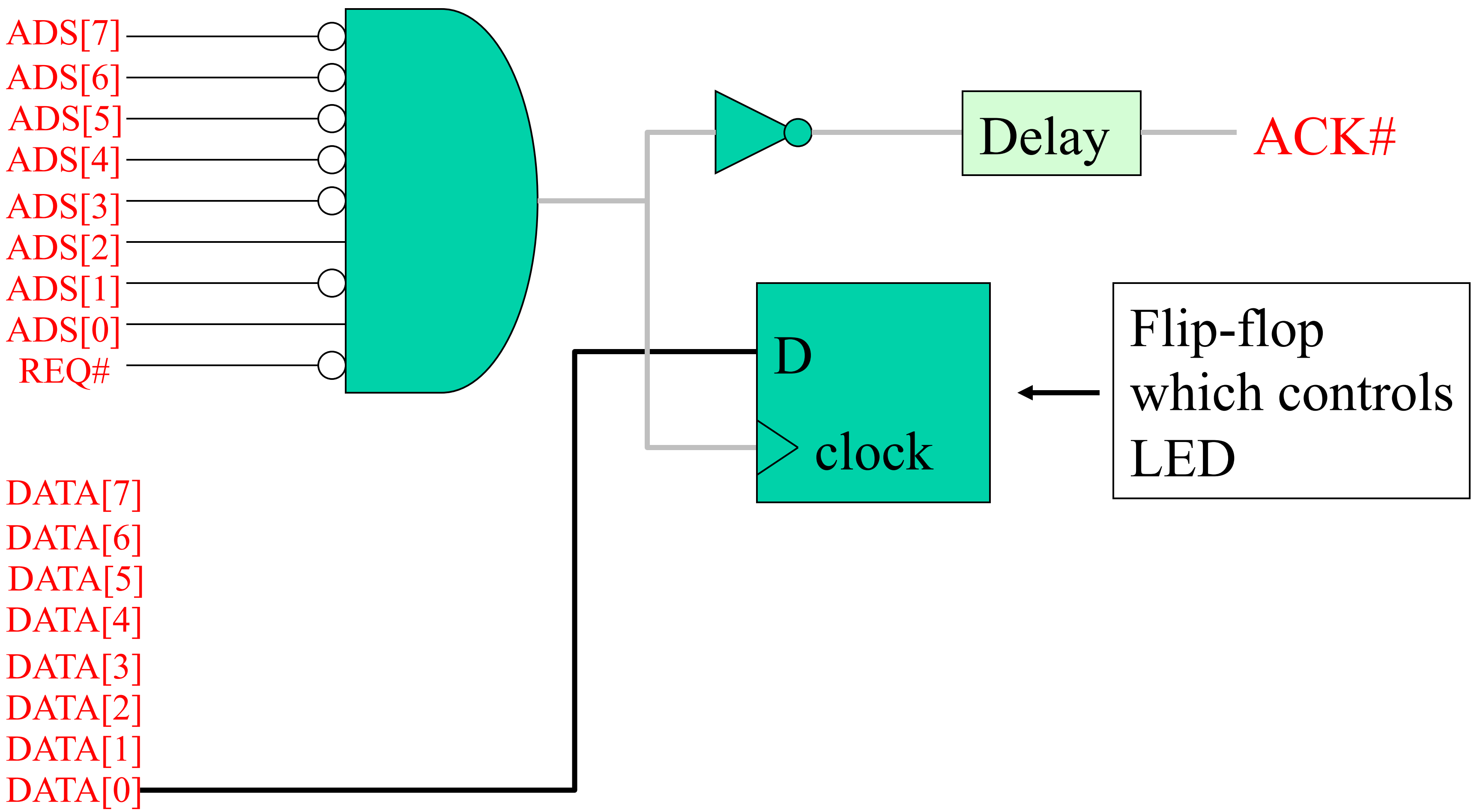
The push-button

(if ADS=0x04 write 0 or 1 depending on button)



The LED

(1 bit reg written by LSB of address 0x05)



Let's write a simple assembly program
Light on if button is pressed, off if not.

What if you wanted it to toggle each time the button was pressed?