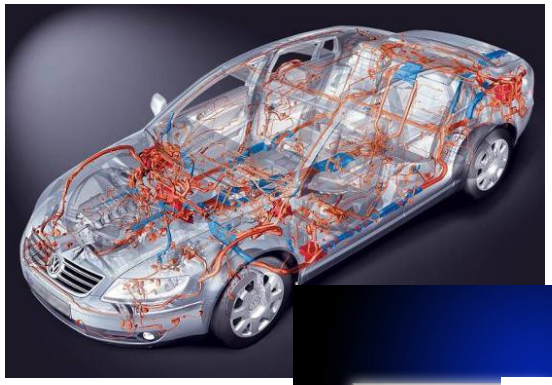
ARM System Introduction

Jim Huang <jserv@0xlab.org>

September 25, 2012

Modifications from Prabal Dutta, University of Michigan

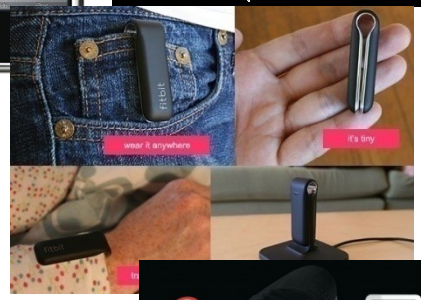
Embedded, everywhere

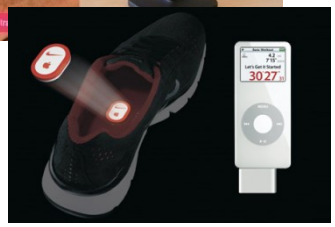


Embedded,
Everywhere

eZ430-Chronos
Wireless Development Tool

Texas Instruments
MSP430





What is driving the
embedded everywhere explosion?

Moore's Law:
IC transistor count doubles every two years

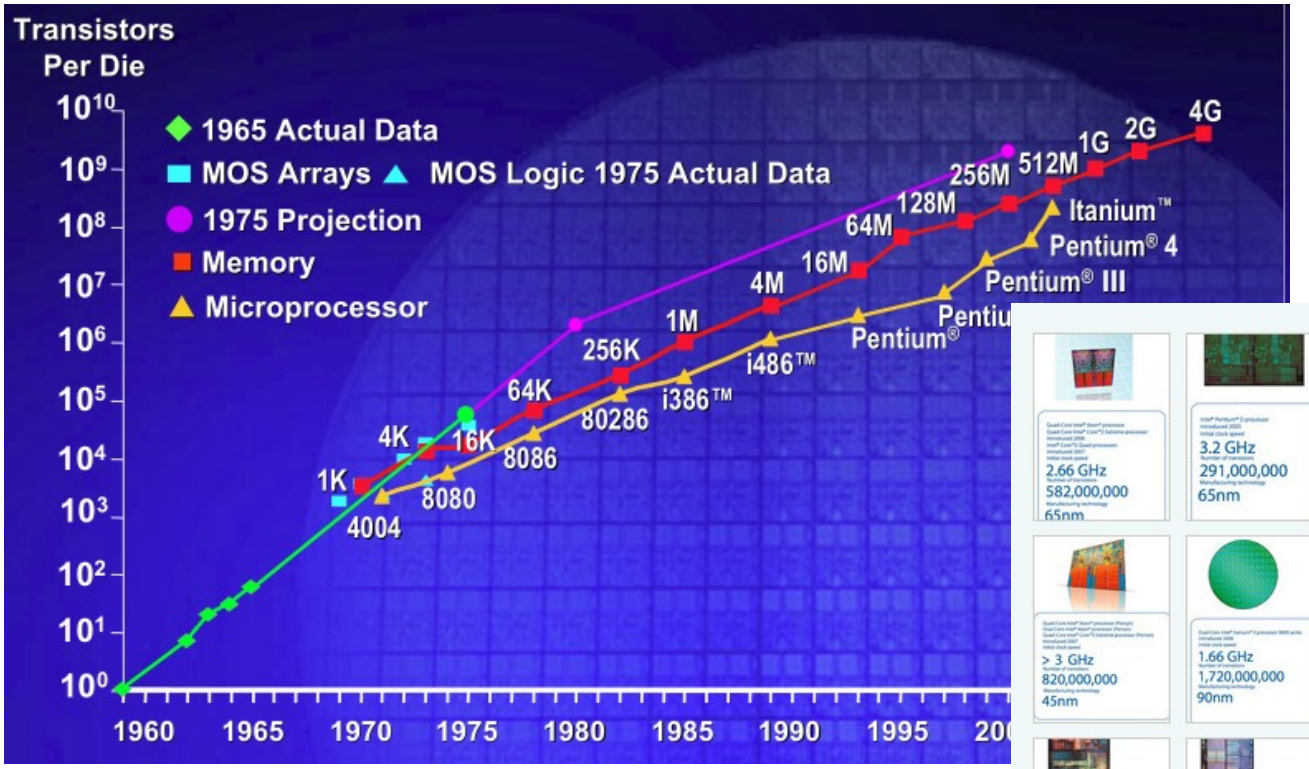
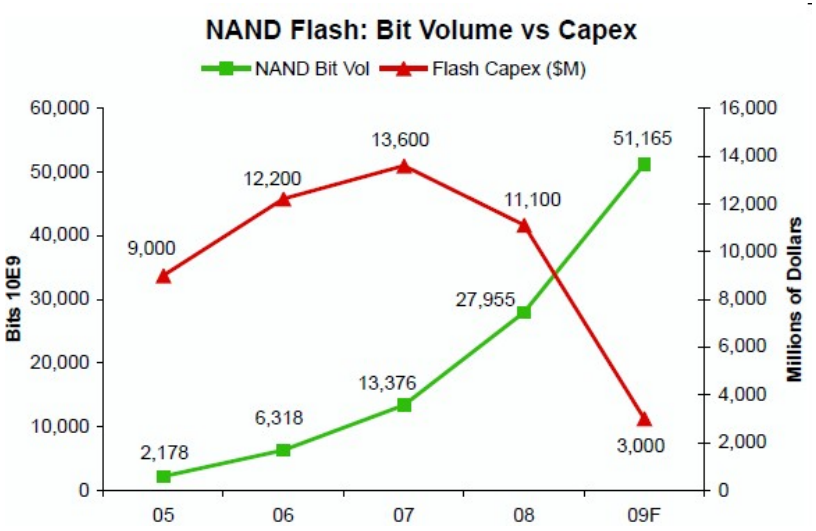


Photo Credit: Intel

Flash memory scaling:
Rise of density & volumes; Fall (and rise) of prices



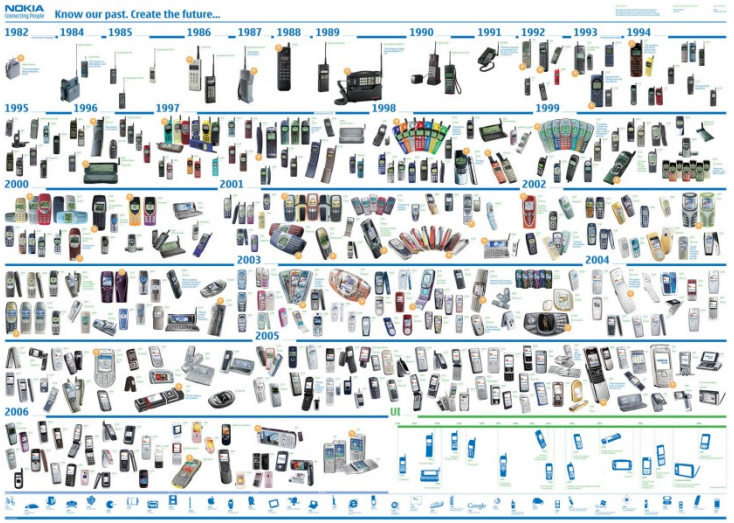
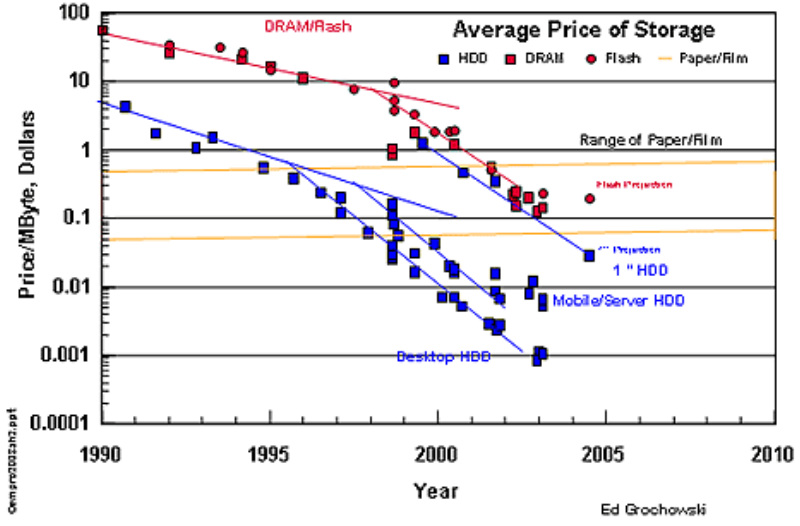
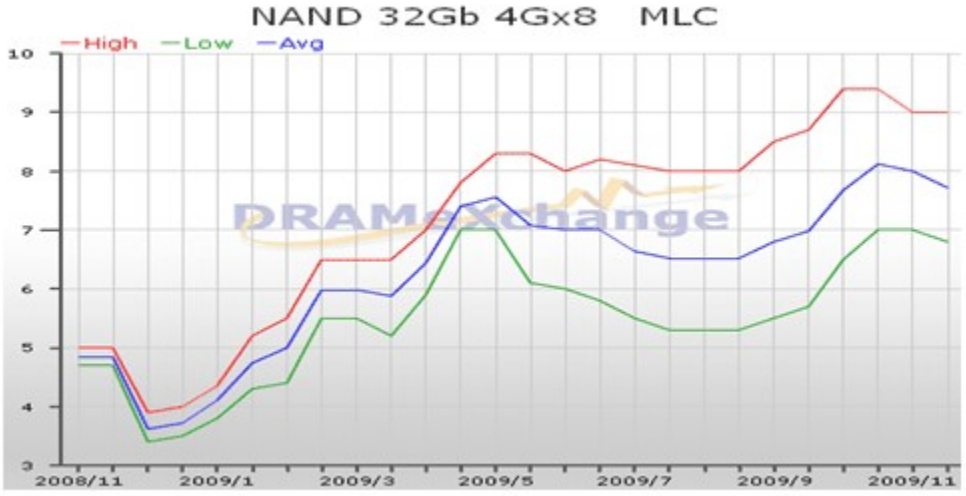


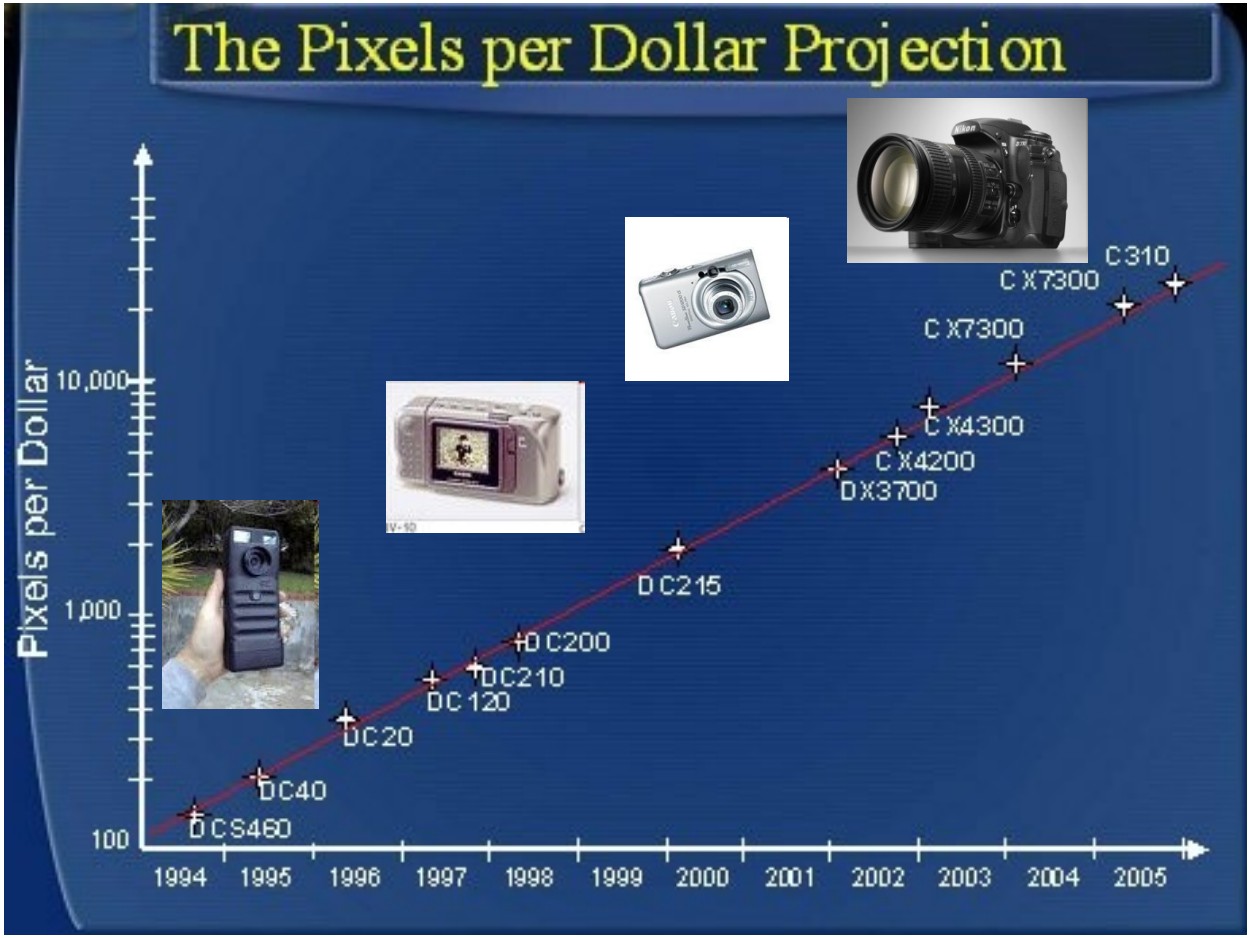
Figure-1 32Gb MLC NAND Flash contract price trend





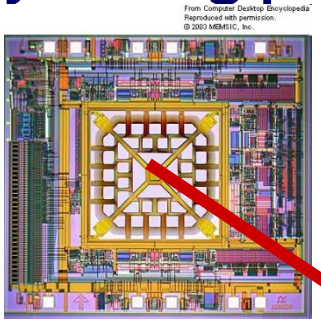
Source: DRAMeXchange, Nov. 2009.

Hendy's "Law": Pixels per dollar doubles annually

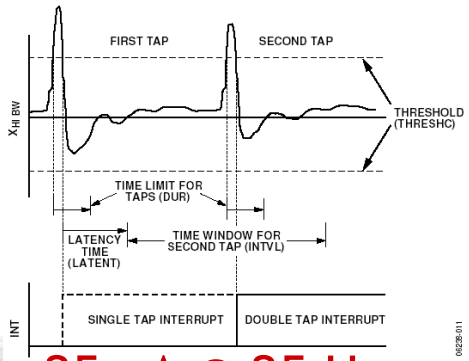


Credit: Barry Hendy/Wikipedia

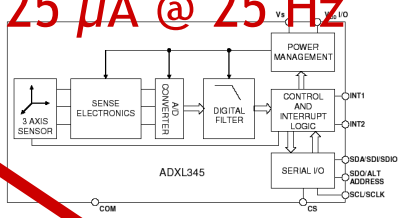
MEMS Accelerometers: Rapidly falling price and power



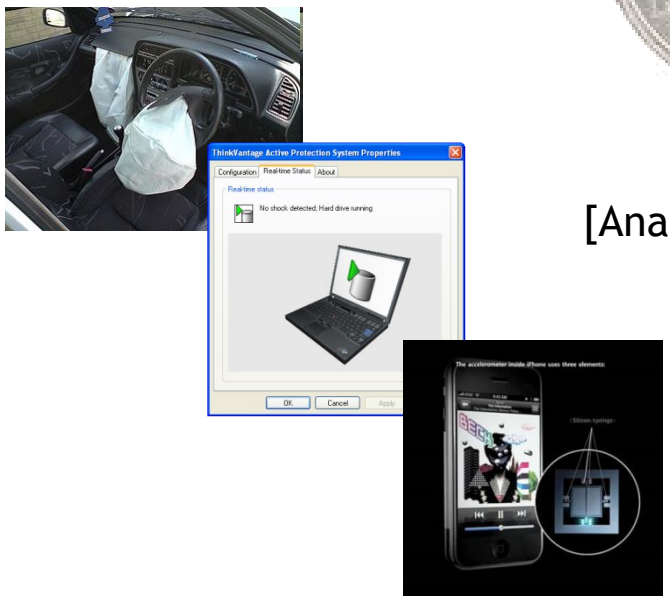
O(mA)



25 μA @ 25 Hz

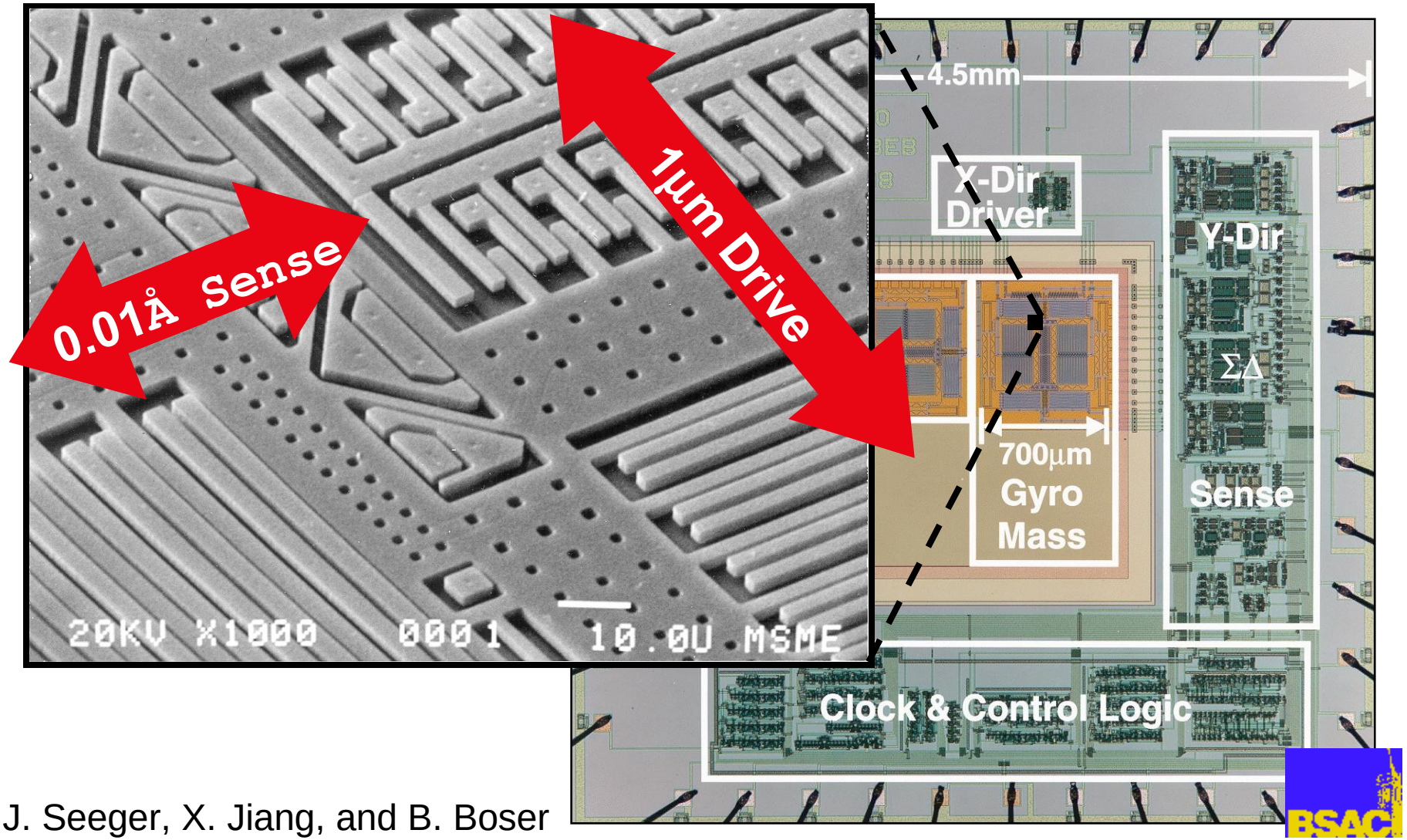


ADXL345
[Analog Devices, 2009]



10 μA @ 10 Hz @ 6 bits
[ST Microelectronics, annc. 2009]

MEMS Gyroscope Chip

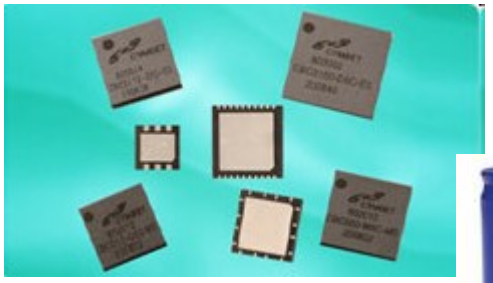


J. Seeger, X. Jiang, and B. Boser

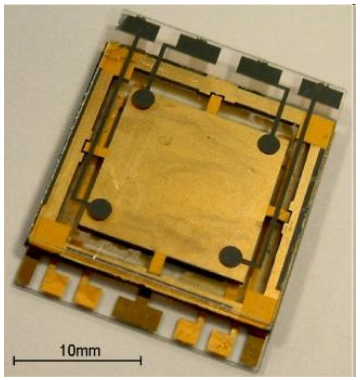
Energy harvesting and storage: Small doesn't mean powerless…

RF [Intel]

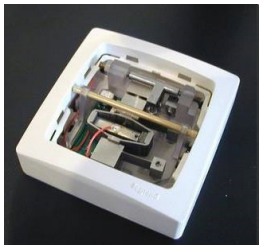
Clare Solar Cell



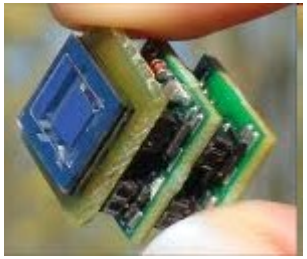
Thin-film batteries



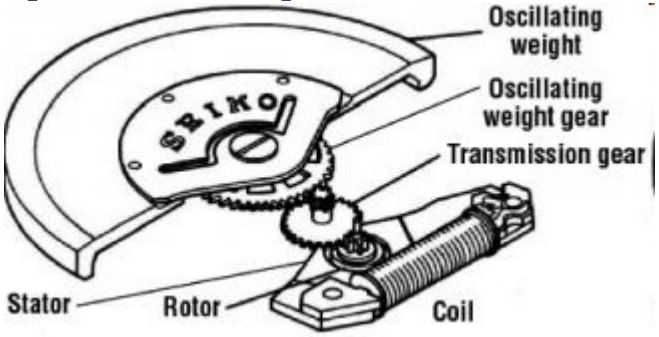
Electrostatic Energy Harvester [ICL]

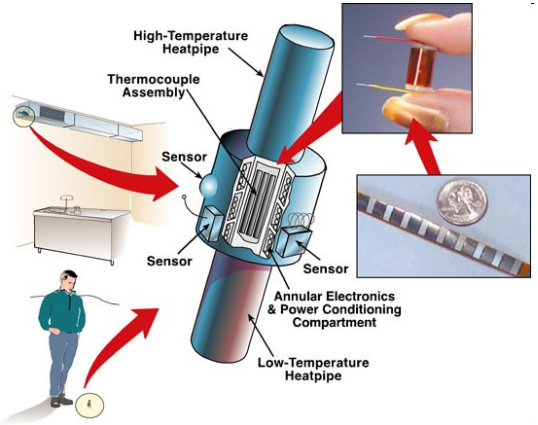


Shock Energy Harvesting CEDRAT Technologies



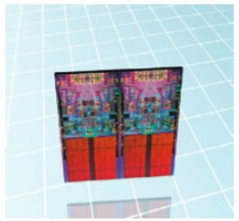
Piezoelectric [Holst/IMEC]

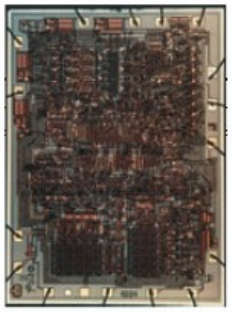




Thermoelectric Ambient Energy Harvester [PNNL]

Bell's Law, Take 2: Corollary to the Laws of Scale

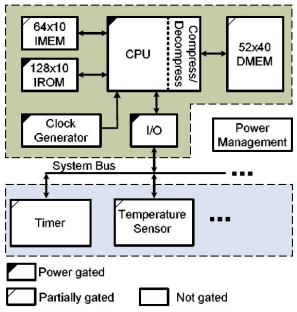


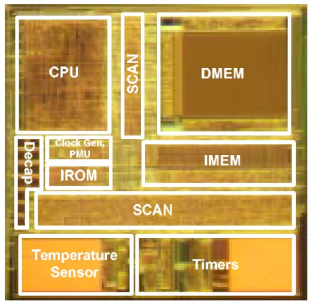


Intel® 4004 processor
Introduced 1971
Initial clock speed
108 KHz
Number of transistors
2,300
Manufacturing technology
10µ

Quad-Core Intel® Xeon® processor
Quad-Core Intel® Core™2 Extreme processor
Introduced 2006
Intel® Core™2 Quad processors
Introduced 2007
Initial clock speed
2.66 GHz
Number of transistors
582,000,000
Manufacturing technology
65nm

15x size decrease
40x transistors
55x smaller λ



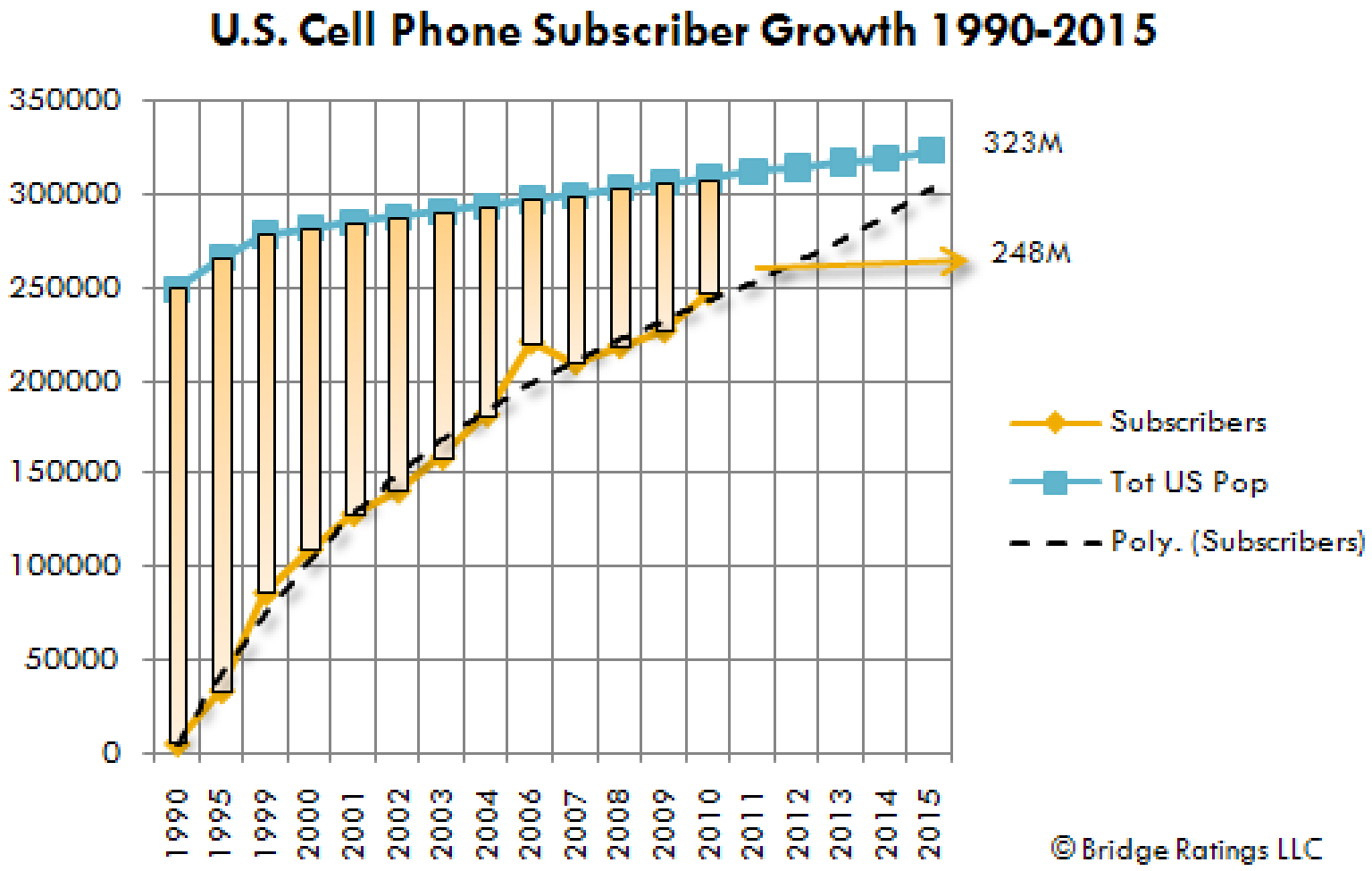


UMich Phoenix Processor
Introduced 2008
Initial clock speed
106 kHz @ 0.5V Vdd
Number of transistors
92,499
Manufacturing technology
0.18 µ

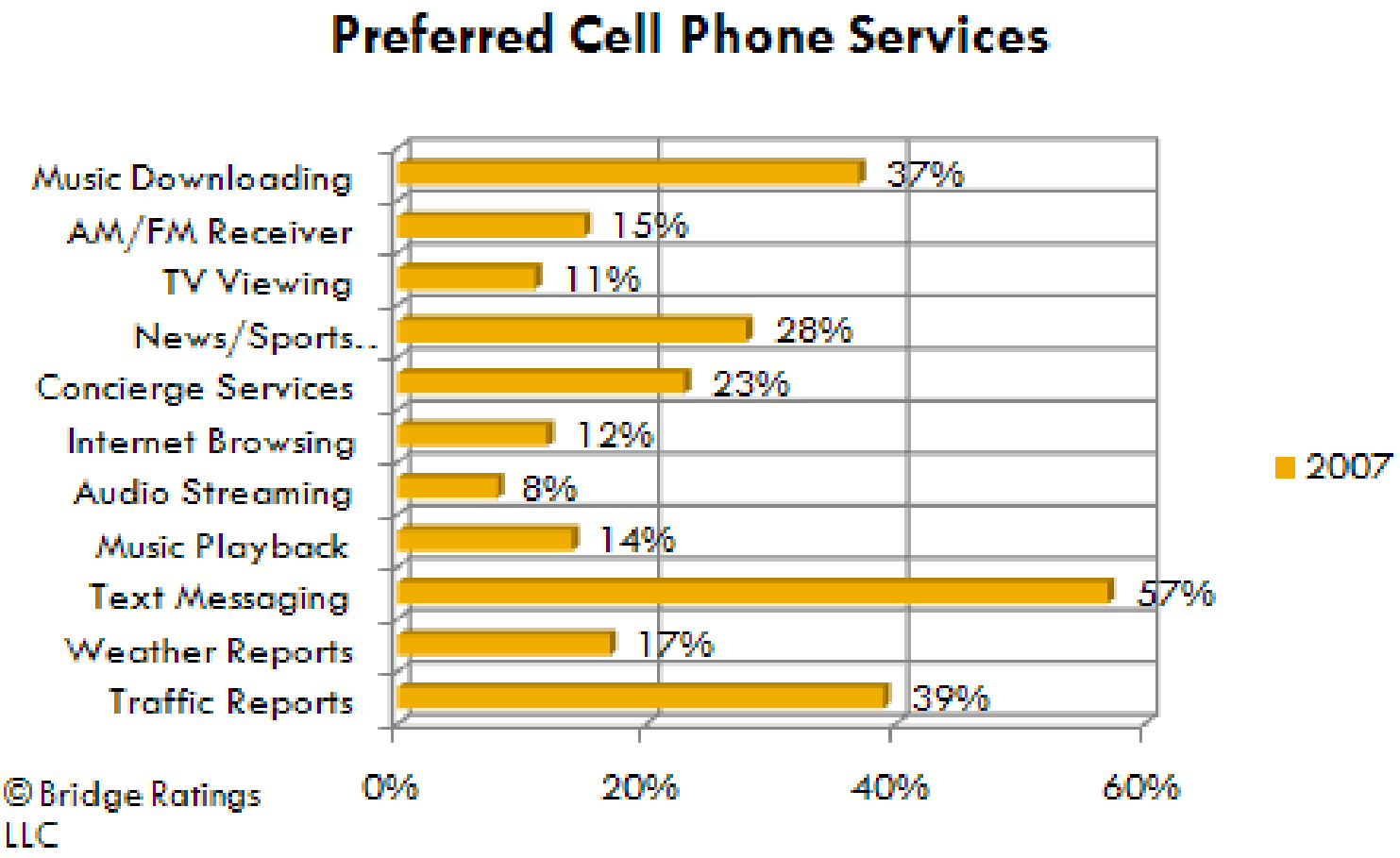
Photo credits: Intel, U. Michigan

Learning happens when assumptions are challenged and invalidated, so…

Mobile phones: the most successful technology ever

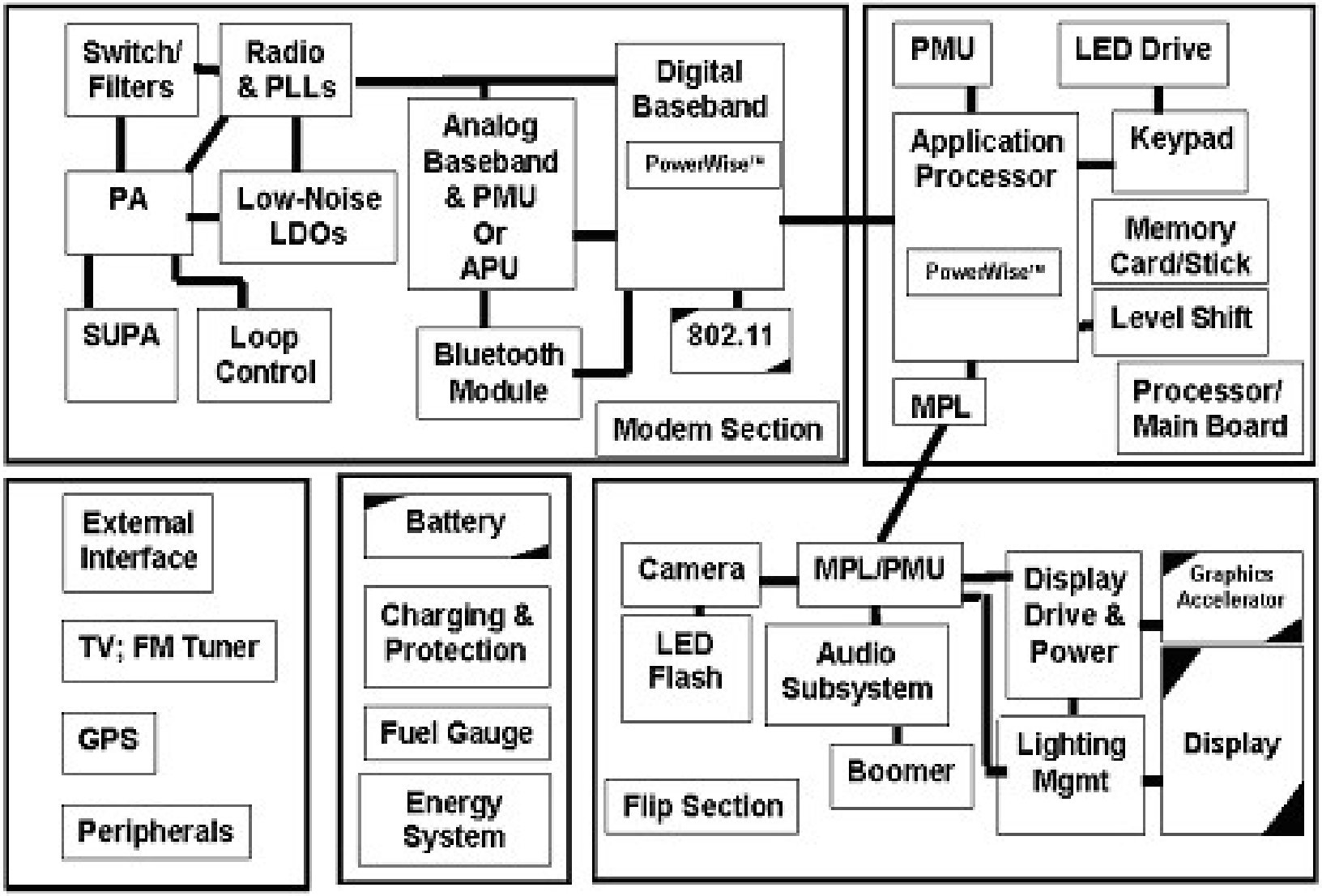


What happened elsewhere now happens on the phone



What happens when you press the power switch on your mobile phone?

Mobile phone system architecture



And, most of them are ARM powered

Lots of manufacturers ship ARM products

Actel
POWER MATTERS

ATMEL

CYPRESS
PERFORM

DUST
NETWORKS

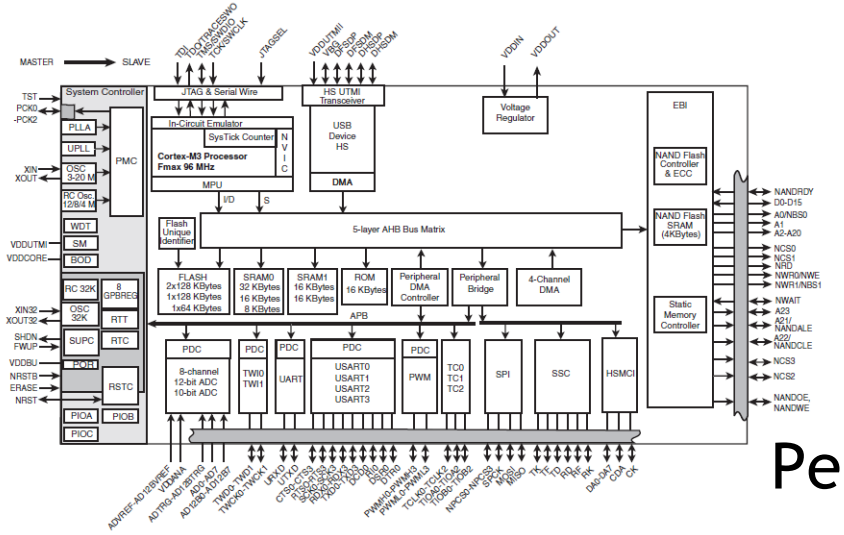
ember

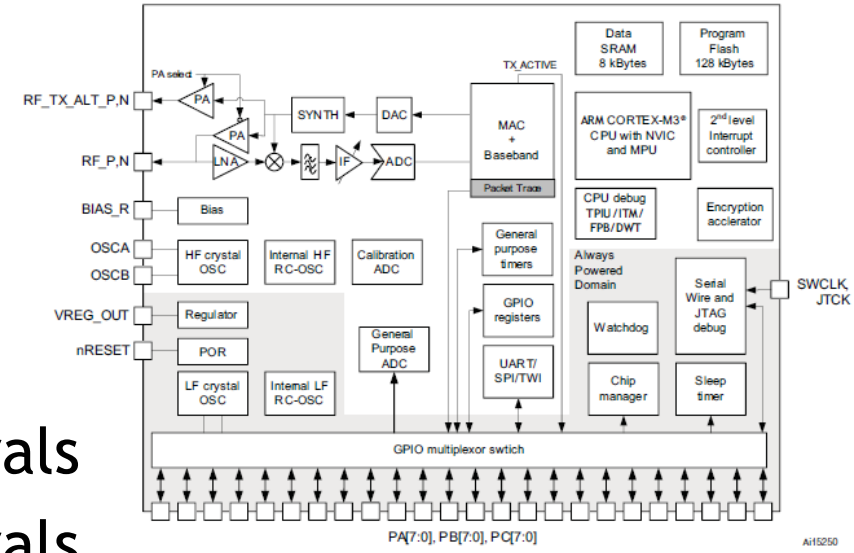
NXP

TEXAS
INSTRUMENTS

What differentiates these products from one another?

The difference is…

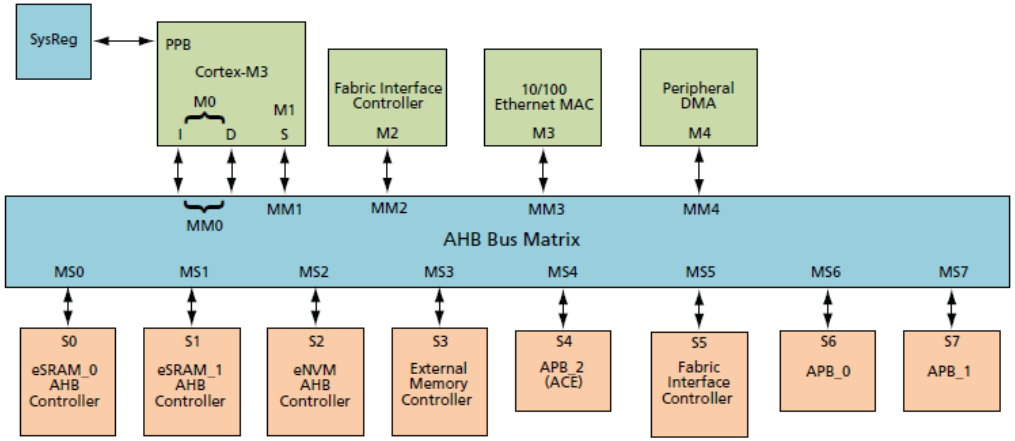




Peripherals

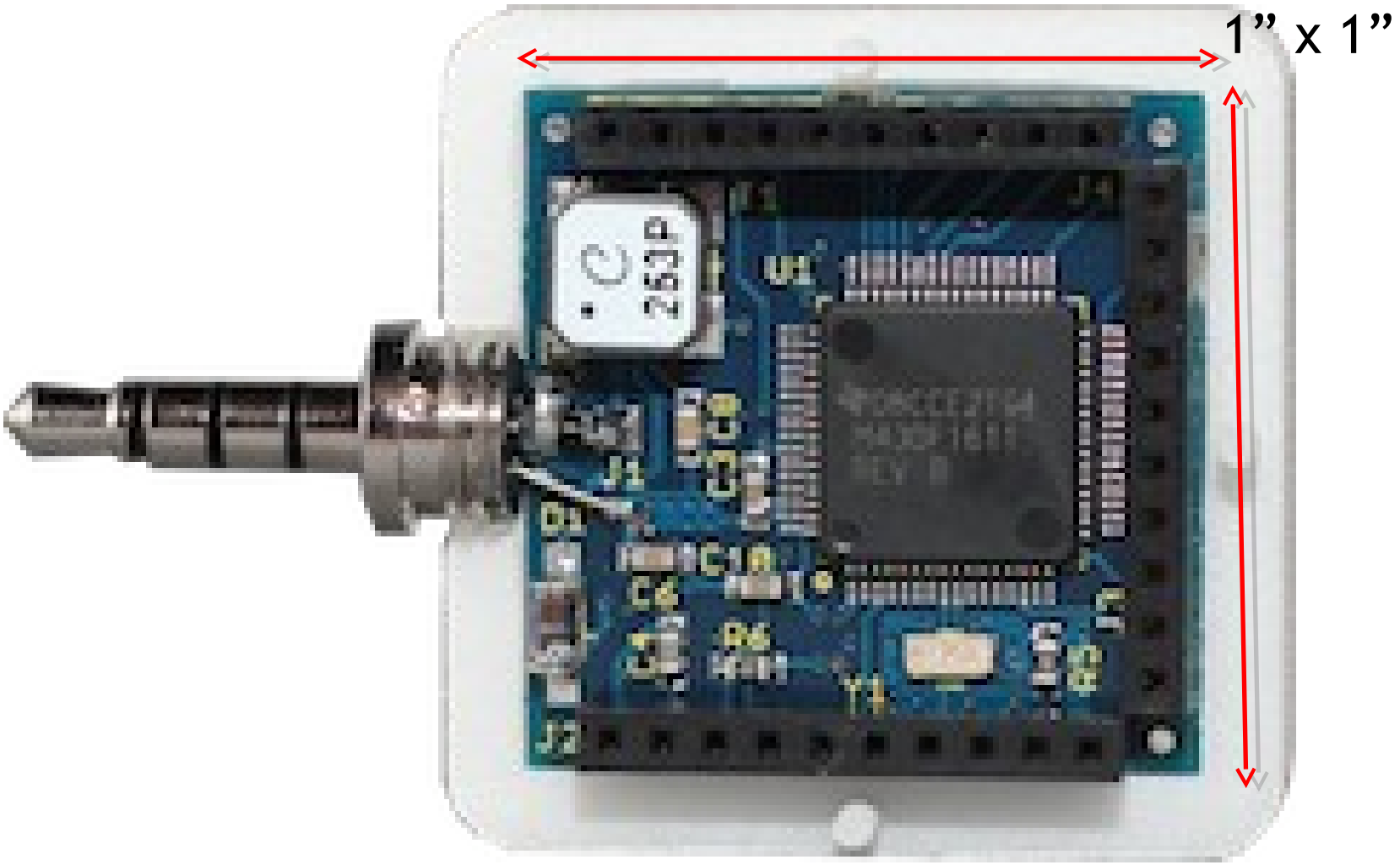
Peripherals

Peripherals

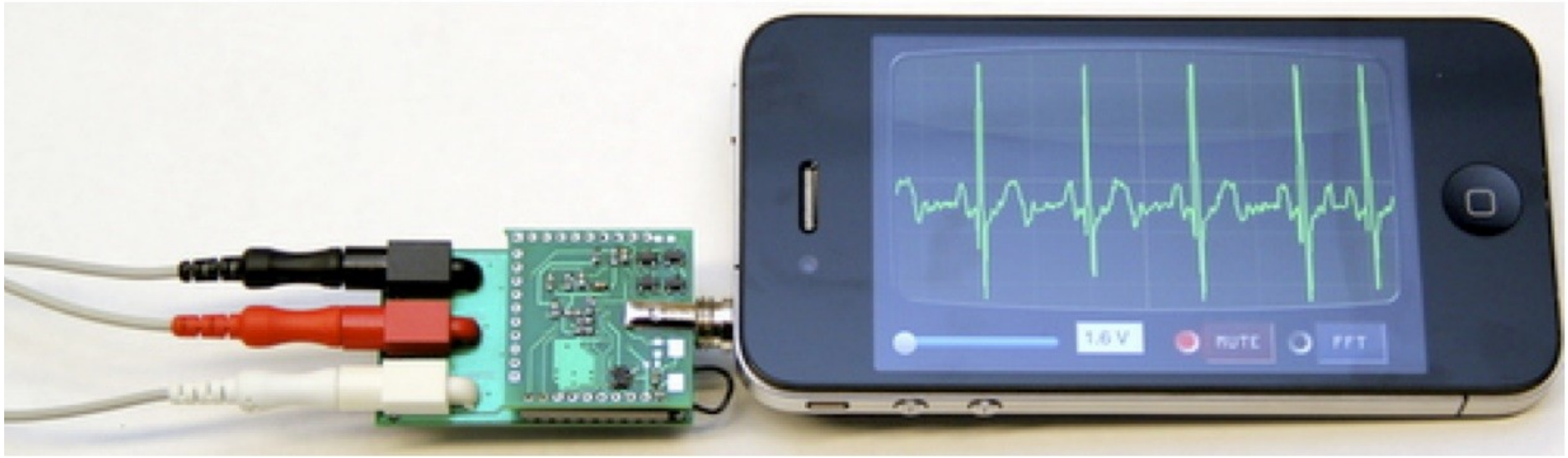


An embedded systems design example: Turning the mobile phone into an EKG station

Integrating power, data, and processing



Integrating power, data, and processing



Architecture

In the context of computers,
what does *architecture* mean?

Architecture has many meanings

- Computer Organization (or Microarchitecture)
  - Control and data paths
  - Pipeline design
  - Cache design
  - ...
- System Design (or Platform Architecture)
  - Memory and I/O buses
  - Memory controllers
  - Direct memory access
  - ...
- Instruction Set Architecture (ISA)

What is an

Instruction Set Architecture (ISA)?

Instruction Set Architecture

"Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"

IBM introducing 360 in 1964

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

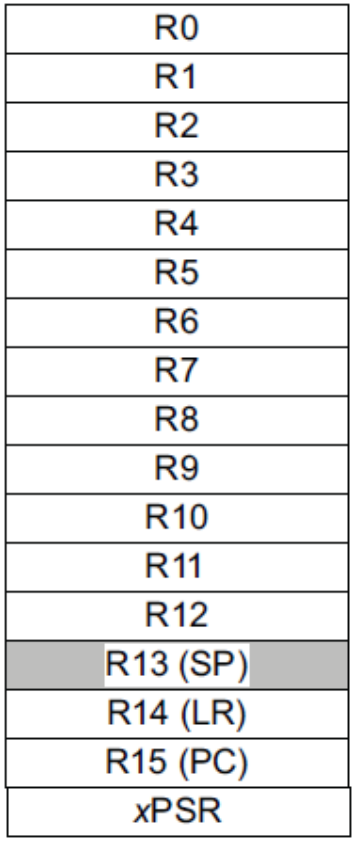
ARM Cortex-M3 ISA

Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

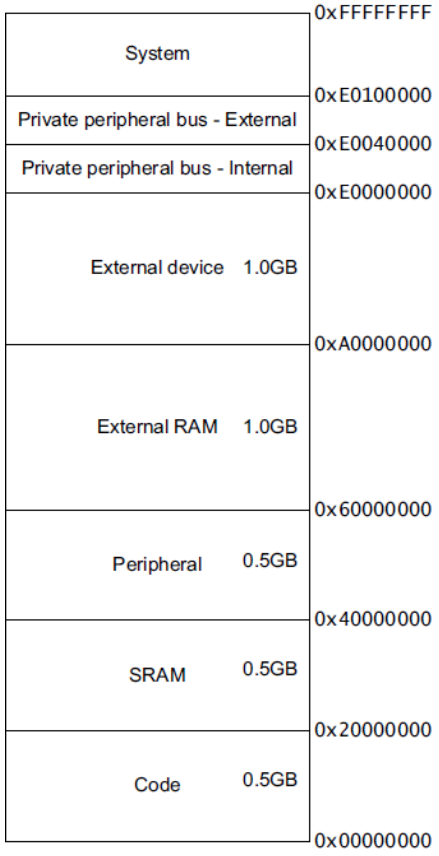
Register Set



32-bits

Endianess

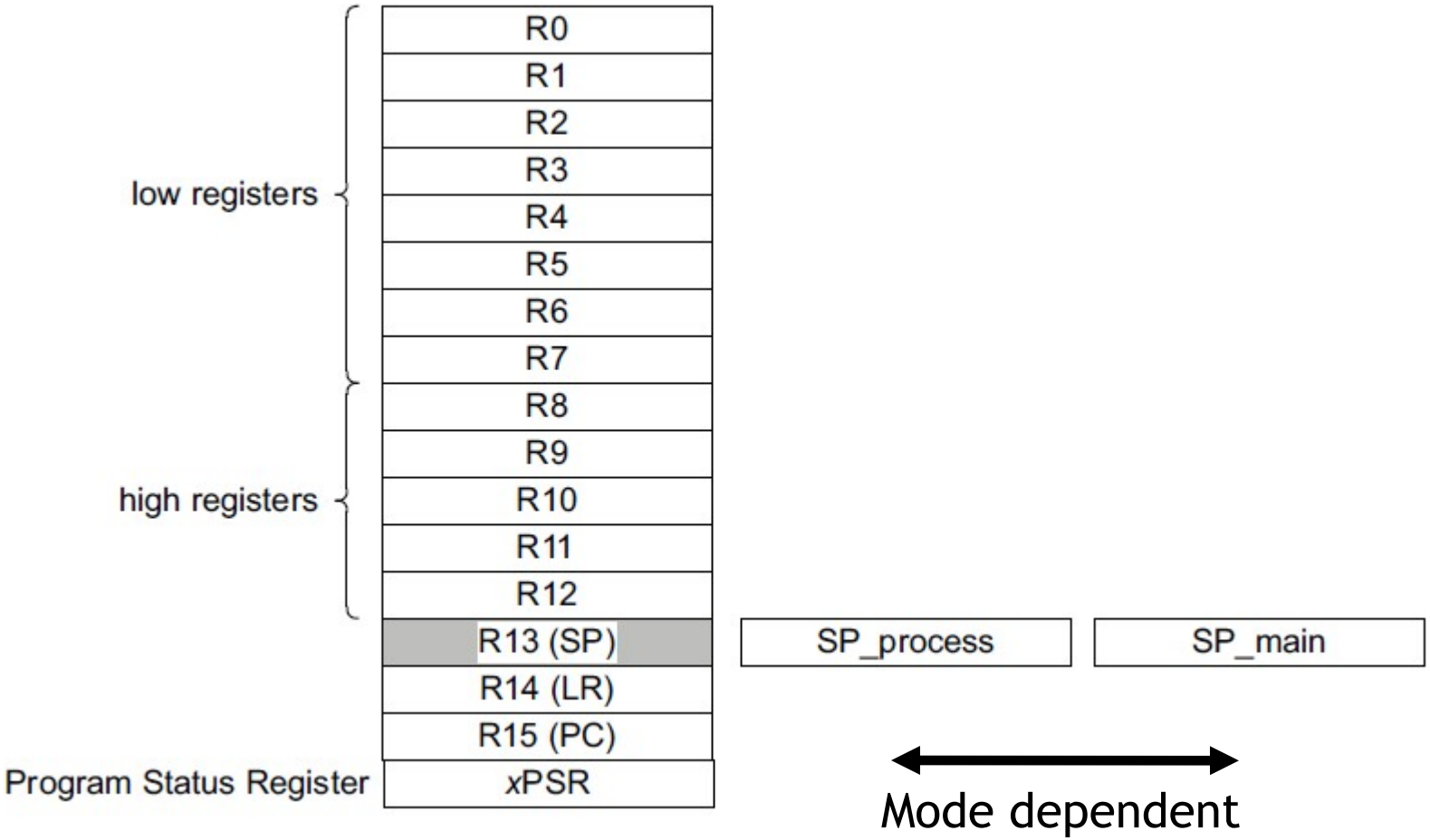
Address Space



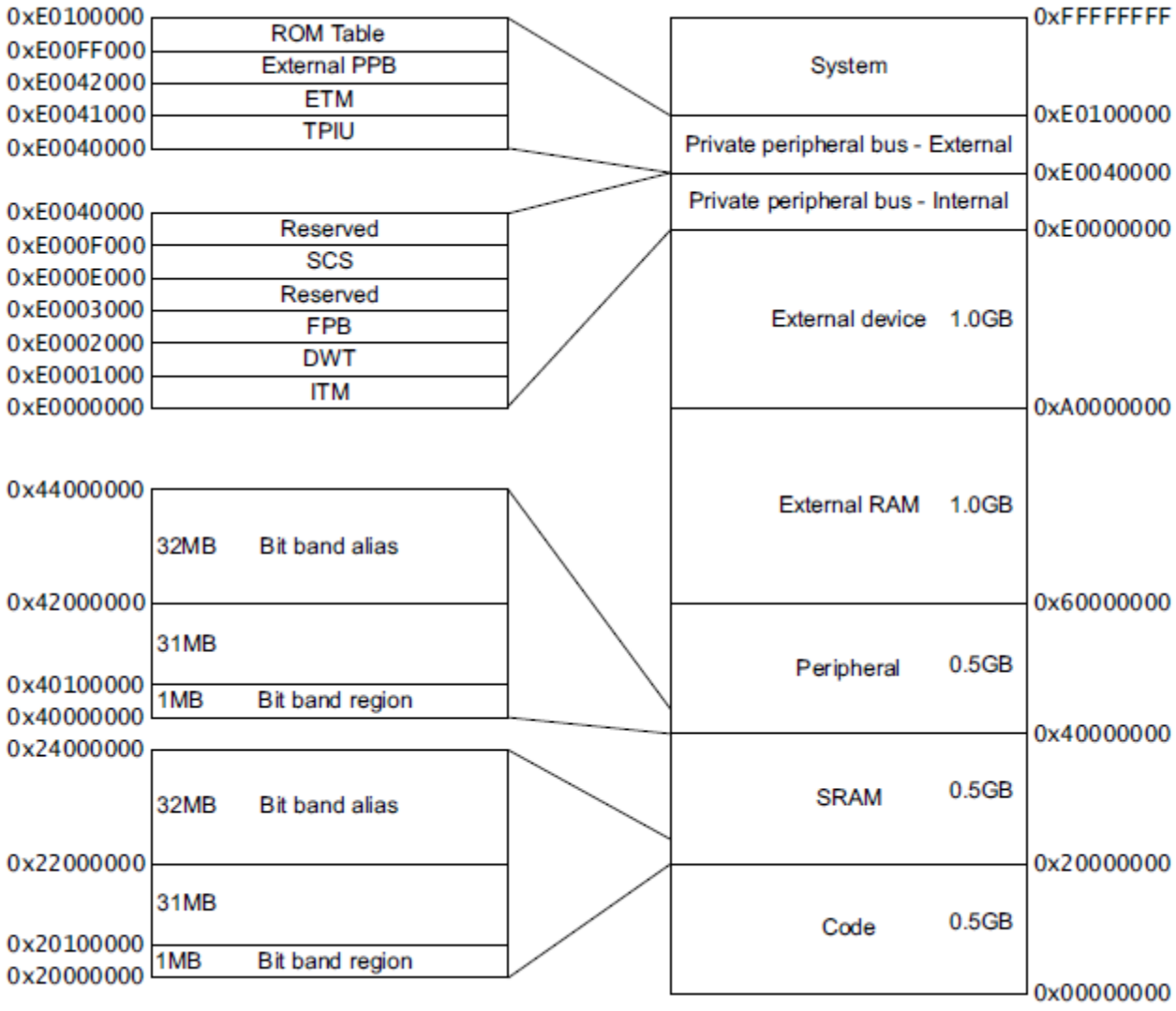
32-bits

Endianess

Registers


low registers
high registers
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
Program Status Register
xPSR
SP_process
SP_main
Mode dependent

Address Space

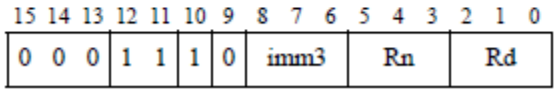


Instruction Encoding
ADD immediate

Encoding T1 All versions of the Thumb ISA.

ADDS <Rd>,<Rn>,#<imm3>

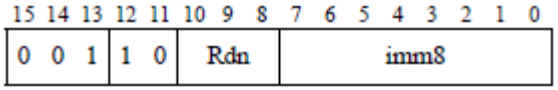
ADD<c> <Rd>,<Rn>,#<imm3>



Encoding T2 All versions of the Thumb ISA.

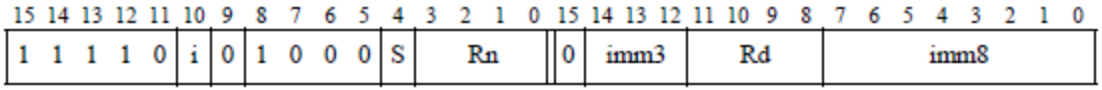
ADDS <Rdn>,#<imm8>

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



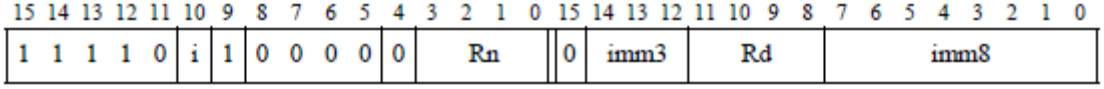
Encoding T3 ARMv7-M

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



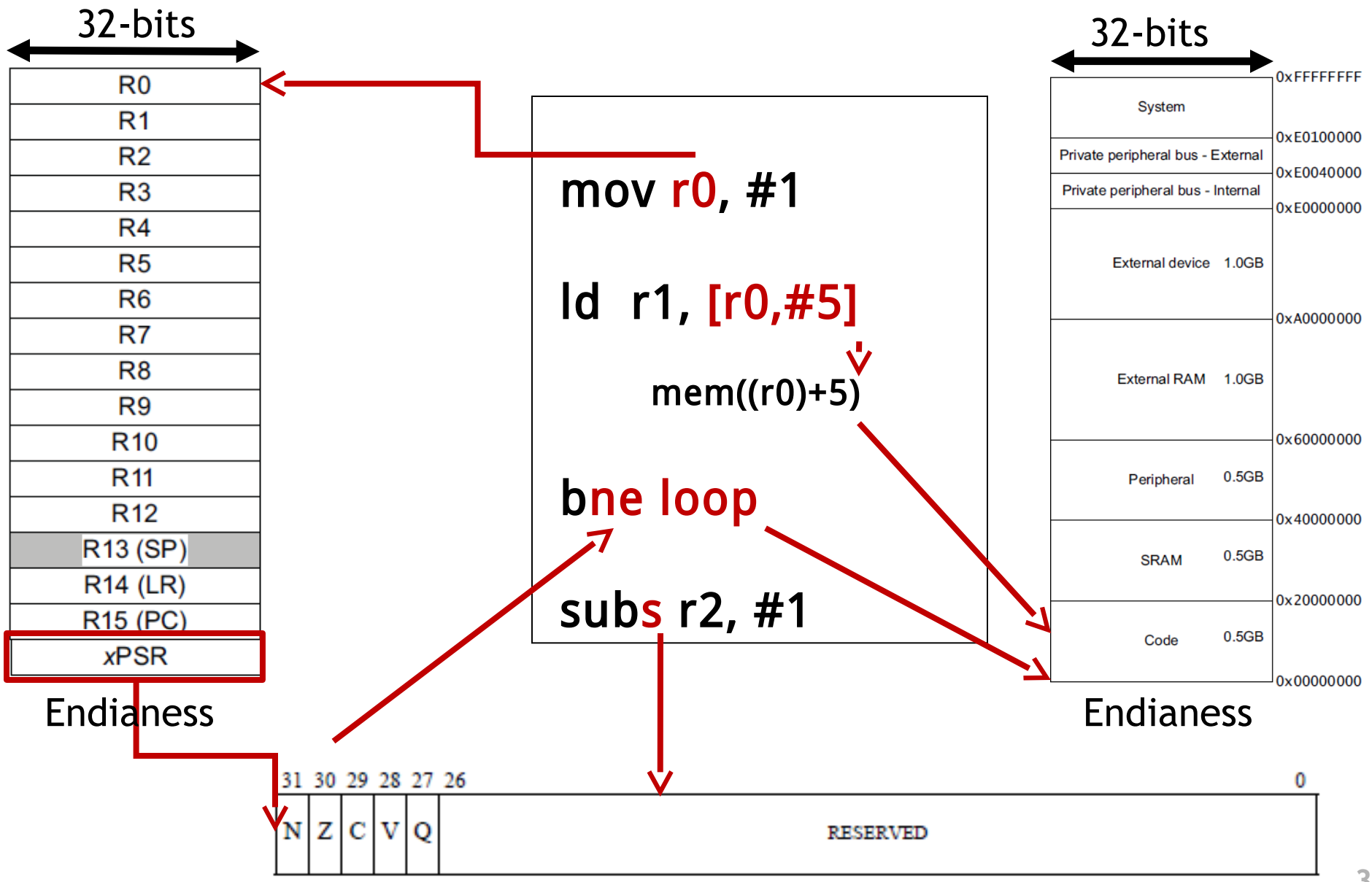
Encoding T4 ARMv7-M

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



Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)



Instruction classes

- Branching
- Data processing
- Load/store
- Exceptions
- Miscellaneous

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>

<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>

Updating the Application Program Status Register (aka condition codes or APRS)

- sub r0, r1
  - r0 ← r0 – r1
  - APSR remain unchanged
- subs r0, r1
  - r0 ← r0 – r1
  - APSR N or Z bits could change

- add r0, r1
  - r0 ← r0 + r1
  - APSR remain unchanged
- adds r0, r1
  - r0 ← r0 + r1
  - APSR C or V bits could change

What does some real assembly look like?

```
0000017c <main>:
     17c:       b580            push    {r7, lr}
     17e:       b084            sub     sp, #16
     180:       af00            add     r7, sp, #0
     182:       f04f 0328       mov.w   r3, #40 ; 0x28
     186:       60bb            str     r3, [r7, #8]
     188:       f04f 0300       mov.w   r3, #0
     18c:       60fb            str     r3, [r7, #12]
     18e:       f04f 0300       mov.w   r3, #0
     192:       603b            str     r3, [r7, #0]
     194:       e010            b.n     1b8 <main+0x3c>
     196:       6838            ldr     r0, [r7, #0]
     198:       f7ff ffb8       bl      10c <factorial>
     19c:       4603            mov     r3, r0
     19e:       607b            str     r3, [r7, #4]
     1a0:       f646 5010       movw    r0, #27920      ; 0x6d10
     1a4:       f2c0 0000       movt    r0, #0
     1a8:       6839            ldr     r1, [r7, #0]
     1aa:       687a            ldr     r2, [r7, #4]
     1ac:       f000 f840       bl      230 <printf>
     1b0:       683b            ldr     r3, [r7, #0]
     ...
```

The endianess religious war: 284 years and counting!

- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- Historical Inspiration
  - Jonathan Swift
  - *Gullivers Travels*
  - Published in 1726
  - Satire on Henry-VIII's split with the Church

- 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
```

Instruction encoding

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

Instructions
movs r0, #10
movs r1, #0

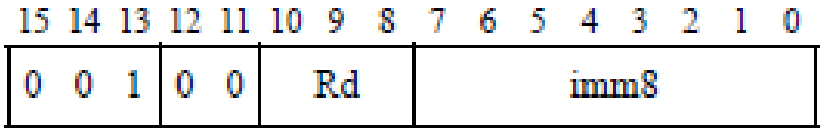
Register Value	Memory Value
001|00|000|00001010 (LSB) (MSB)	
(msb) (lsb) 0a 20 00 21	
001|00|001|00000000	

ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

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

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

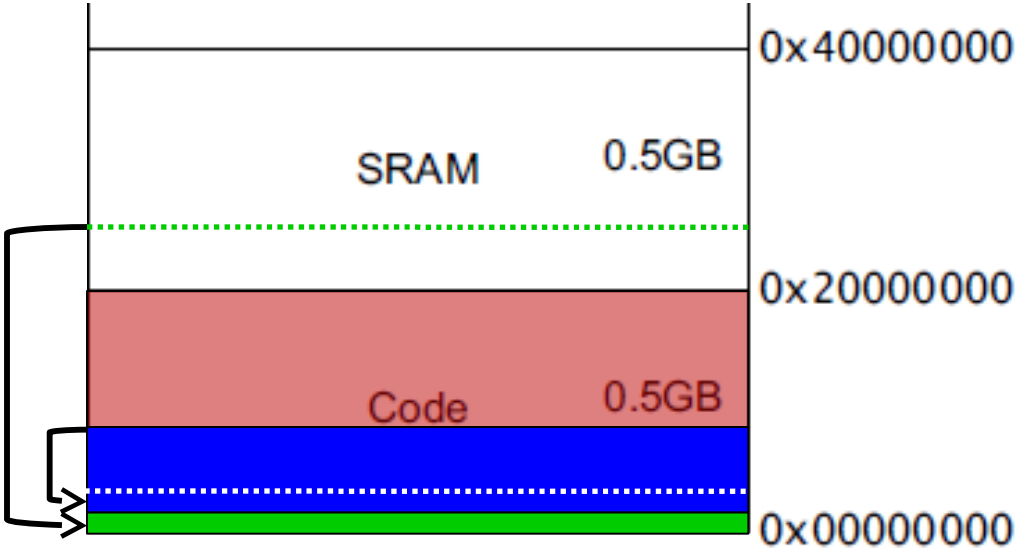



```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

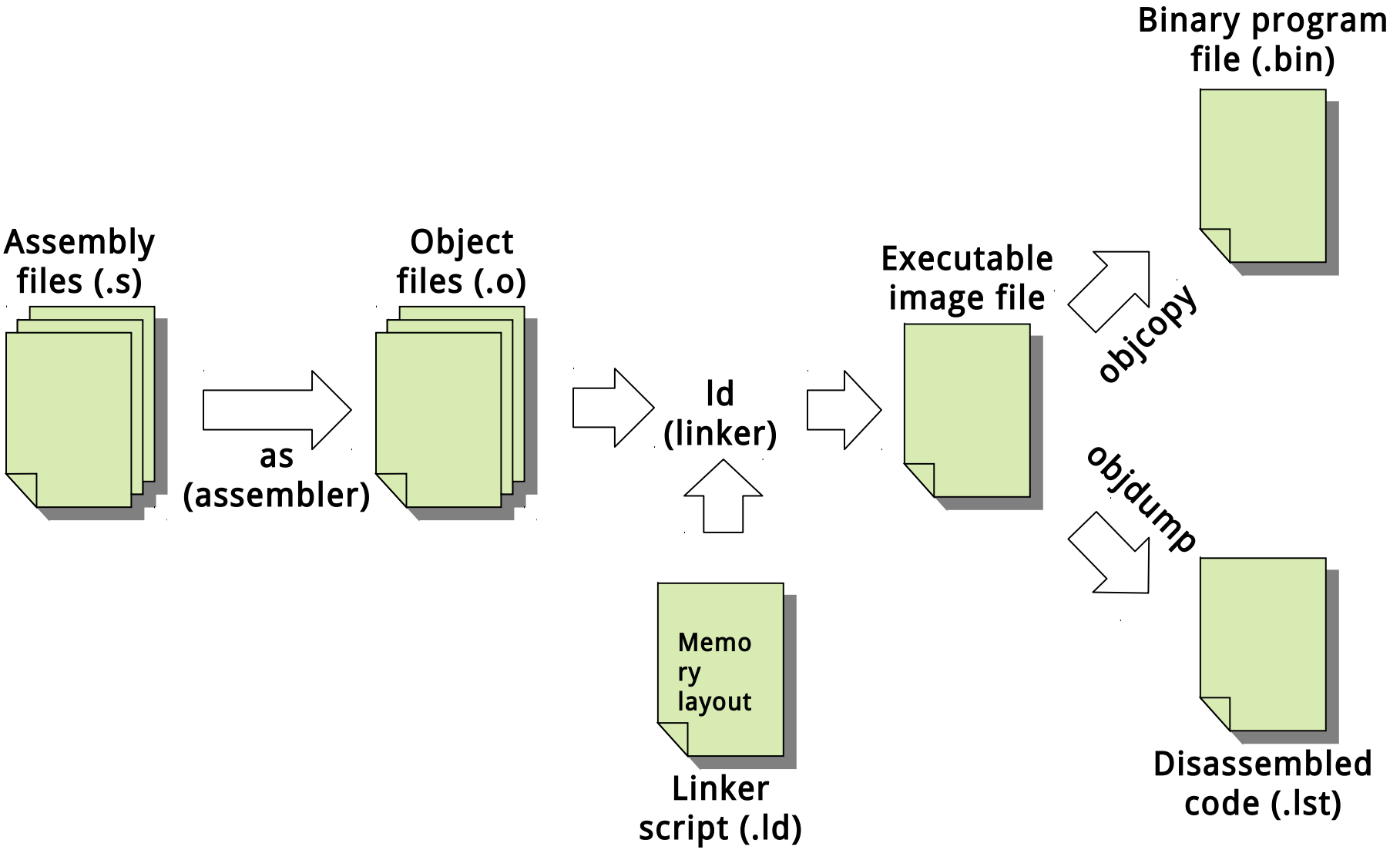
What happens after a power-on-reset (POR)?

- On the ARM Cortex-M3
- SP and PC are loaded from the code (.text) segment
- Initial stack pointer
  - LOC: 0x00000000
  - POR: SP ← mem(0x00000000)
- Interrupt vector table
  - *Initial* base: 0x00000004
  - Vector table is relocatable
  - Entries: 32-bit values
  - Each entry is an address
  - Entry #1: reset vector
    - LOC: 0x0000004
    - POR: PC ← mem(0x00000004)
- Execution begins

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

_start:

        .word   STACK_TOP, start

start:
        movs r0, #10
        ...
```



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++

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>
```

What are the elements of a real assembly program?

```
        .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
```

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

How can the contents of an object file be read?

- $ readelf –a example.o

- Shows
  - ELF headers
  - Program headers
  - Section headers
  - Symbol table
  - Files attributes

- Other options
  - -s shows symbols
  - -S shows section headers

What does an object file contain?

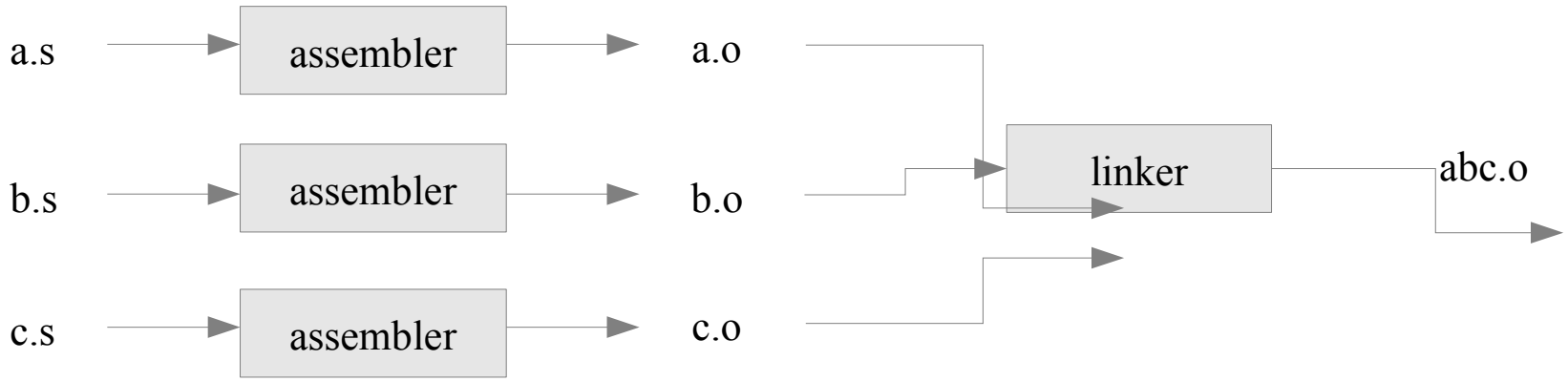
```
$ readelf -S example1.o
There are 9 section headers, starting at offset 0xac:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000014 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 000300 000008 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000048 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000048 000000 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000048 000021 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000069 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 000214 0000c0 10      8  11  4
  [ 8] .strtab           STRTAB          00000000 0002d4 00002c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

How are object files linked?

- $ arm-none-eabi-ld
  - Useful options
    - -Ttext
    - -Tbss
    - -o

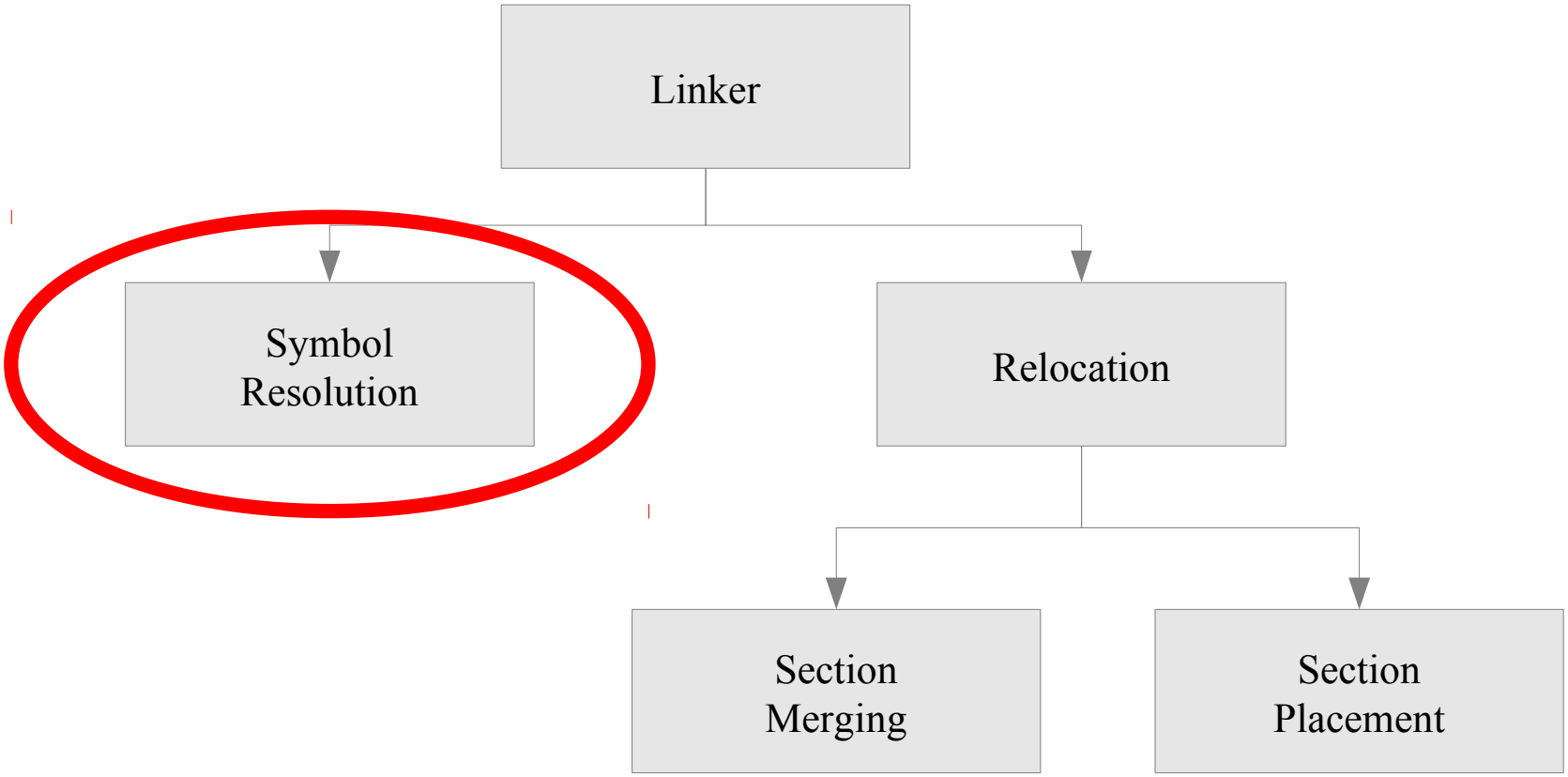
$ arm-none-eabi-ld -Ttext 0x0 –Tbss 0x20000000 -o example1.out example1.o

Linker



- In multi-file programs – combines multiple object files to form executable

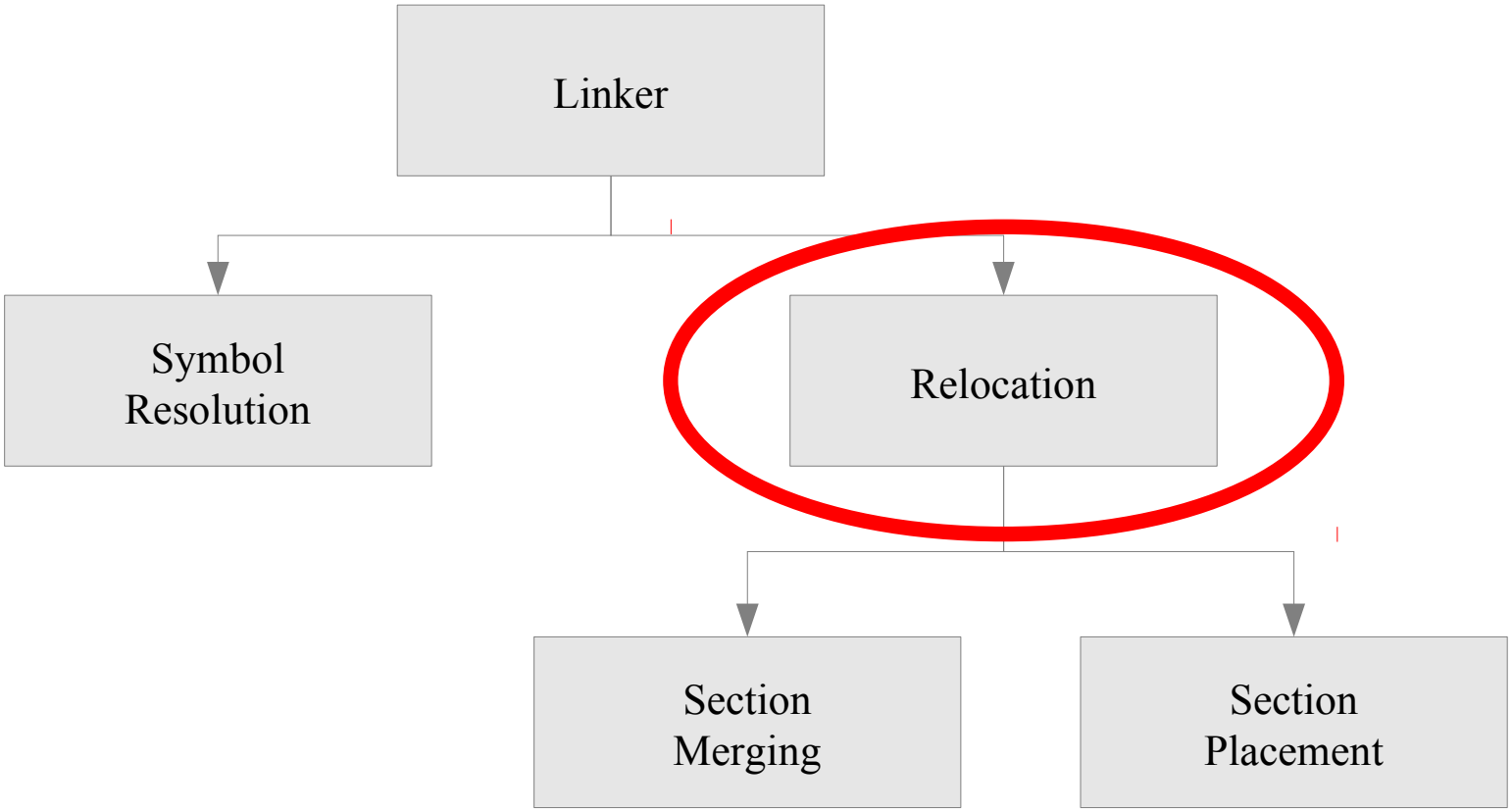
Linker


Linker
Symbol Resolution
Relocation
Section Merging
Section Placement

Symbol Resolution

- Functions are defined in one file
- Referenced in another file
- References are marked unresolved by the compiler
- Linker patches the references

Linker


Linker
Symbol Resolution
Relocation
Section Merging
Section Placement

Relocation

- Code generated assuming it starts from address X
- Code should start from address Y
- Change addresses assigned to labels
- Patch label references

Sections

- Placing related bytes at a particular location.
- Example:
  - instructions in Flash
  - data in RAM
- Related bytes are grouped together using sections
- Placement of sections can be specified

Sections

- Most programs have atleast two sections, .text and .data
- Data or instructions can be placed in a section using directives
- Directives
  - .text
  - .data
  - .section

Sections

```
        .data
arr:    .word 10, 20, 30, 40, 50
len:    .word 5
        .text
start:  mov r1, #10
        mov r2, #20
        .data
result: .skip 4
        .text
        add r3, r2, r1
        sub r3, r2, r1
```

.data section

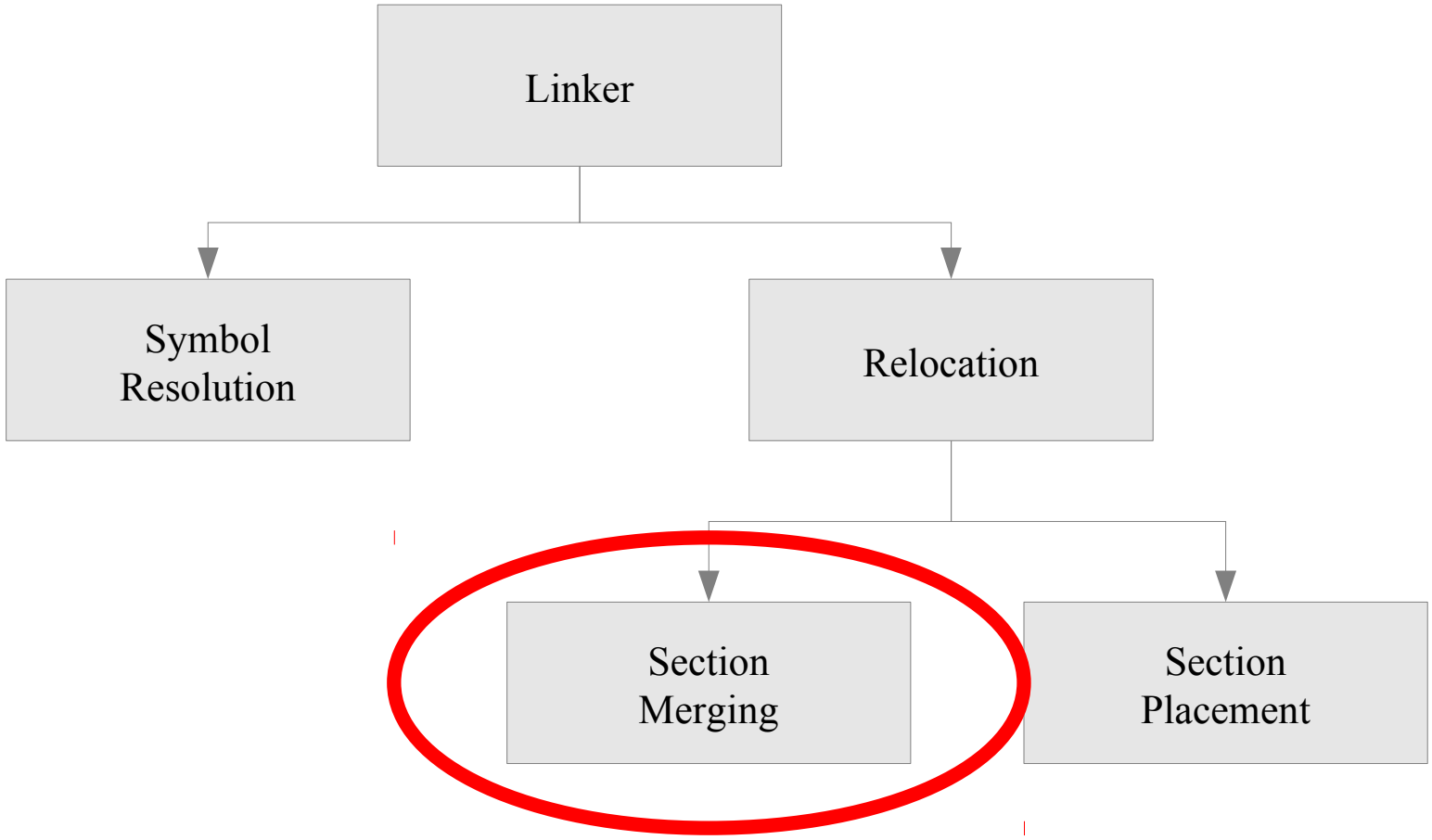
```
0000_0000 arr:    .word 10, 20, 30, 40, 50
0000_0014 len:    .word 5
0000_0018 result: .skip 4
```

.text section

```
0000_0000 start:  mov r1, #10
0000_0004         mov r2, #20
0000_0008         add r3, r2, r1
0000_000C         sub r3, r2, r1
```

- Source – sections can be interleaved
- Bytes of a section – contiguous addresses

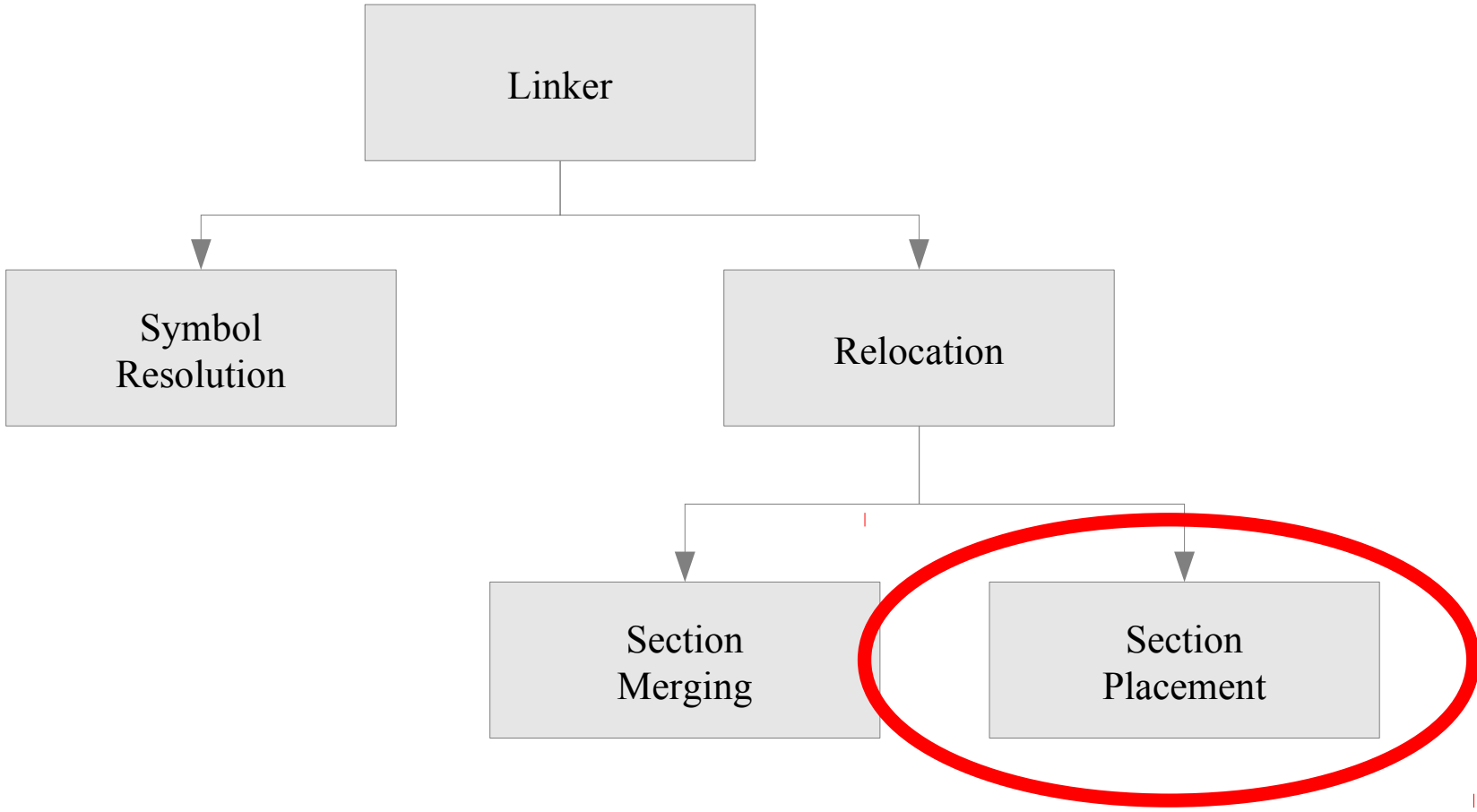
Linker


Linker
Symbol Resolution
Relocation
Section Merging
Section Placement

Section Merging

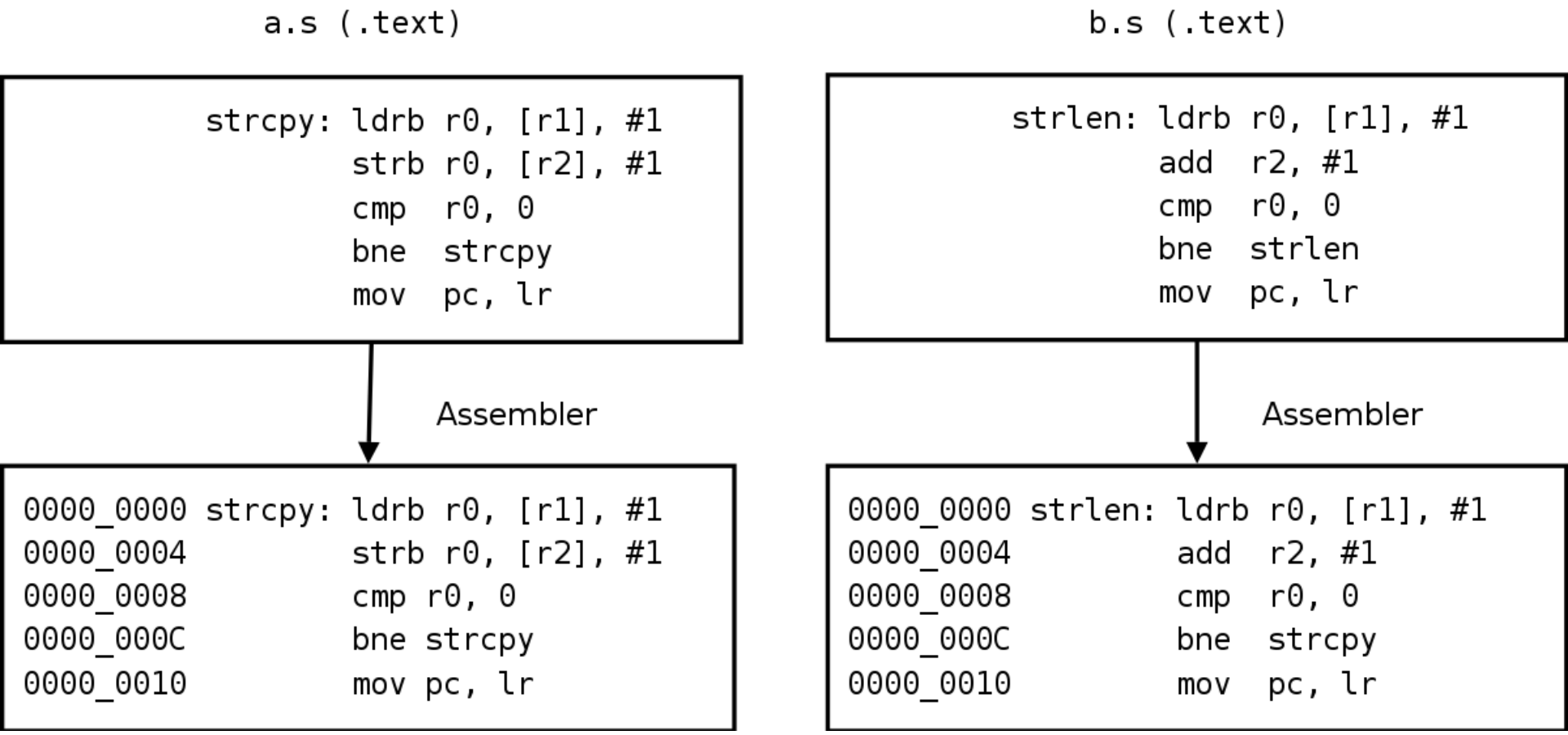
- Linker merges sections in the input files into sections in the output file
- Default merging – sections of same name
- Symbols get new addresses, and references are patched
- Section merging can be controlled by linker script files

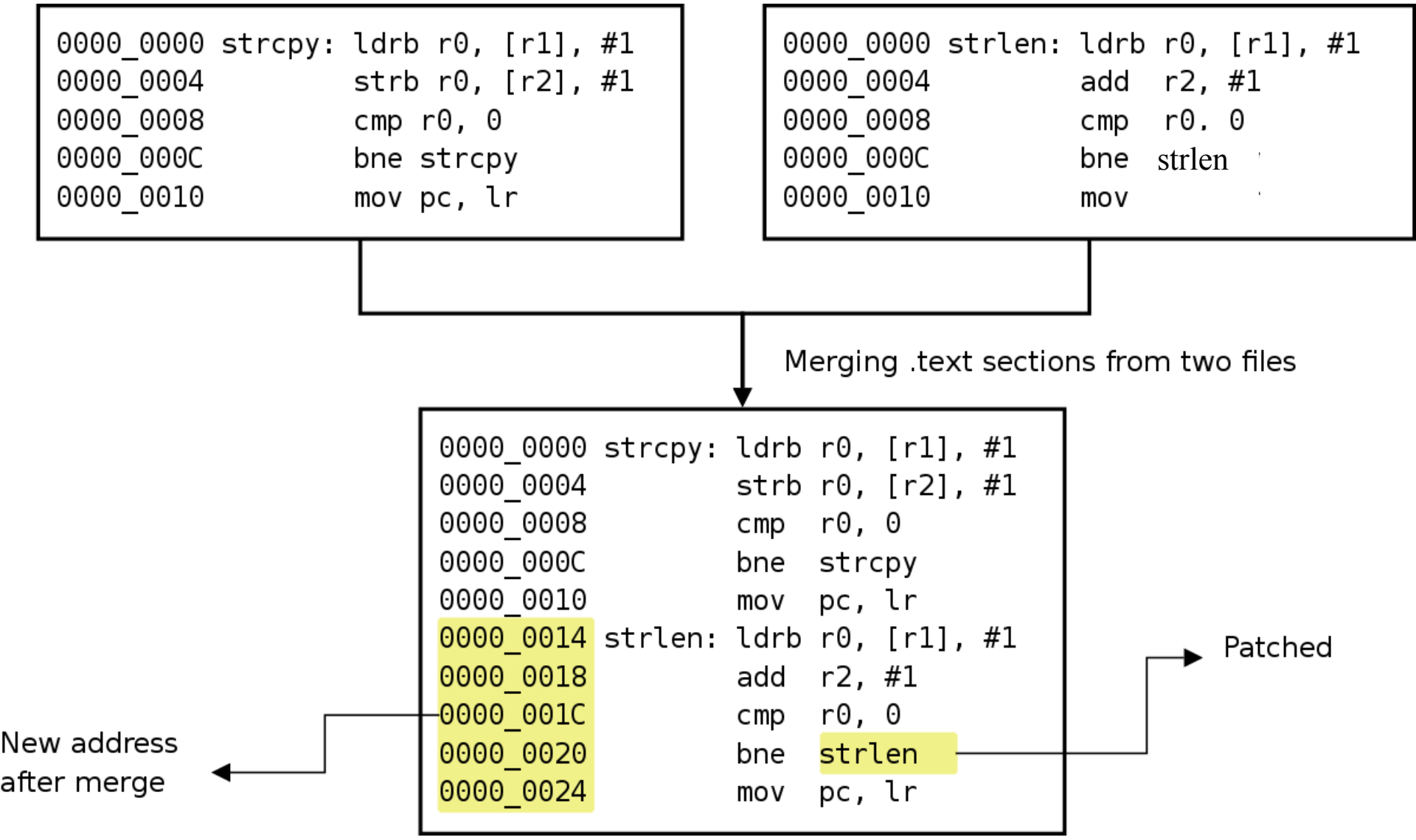
Linker

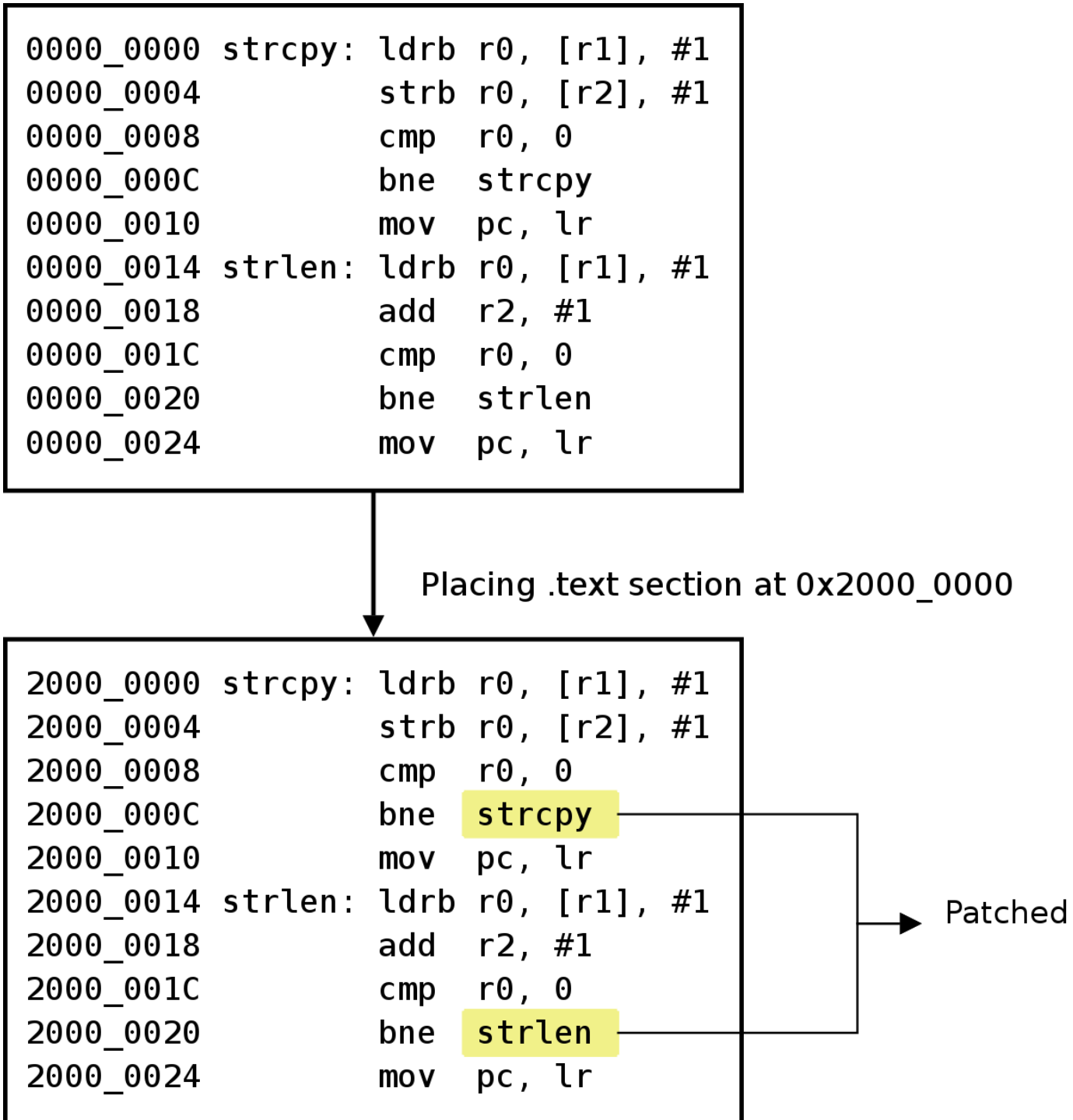

Linker
Symbol Resolution
Relocation
Section Merging
Section Placement

Section Placement

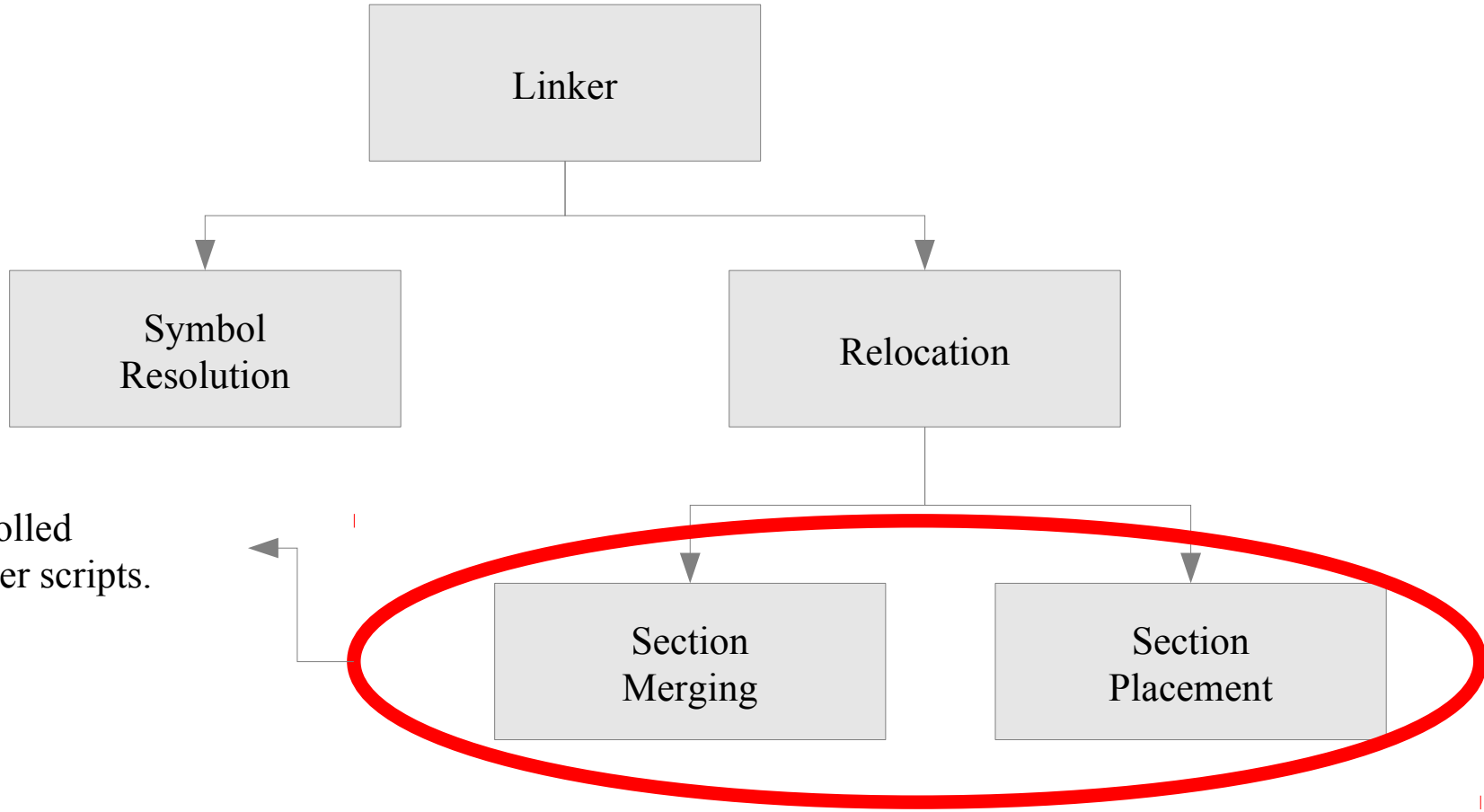
- Bytes in each section is given addresses starting from 0x0
- Labels get addresses relative to the start of section
- Linker places section at a particular address
- Labels get new address, label references are patched


a.s (.text)
strcpy: ldrb r0, [r1], #1
strb r0, [r2], #1
cmp r0, 0
bne strcpy
mov pc, lr
Assembler
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
b.s (.text)
strlen: ldrb r0, [r1], #1
add r2, #1
cmp r0, 0
bne strlen
mov pc, lr
Assembler
0000_0000 strlen: ldrb r0, [r1], #1
0000_0004 add r2, #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr


0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0000 strlen: ldrb r0, [r1], #1
0000_0004 add r2, #1
0000_0008 cmp r0. 0
0000_000C bne strlen
0000_0010 mov
Merging .text sections from two files
0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0014 strlen: ldrb r0, [r1], #1
0000_0018 add r2, #1
0000_001C cmp r0, 0
0000_0020 bne strlen
0000_0024 mov pc, lr
Patched
New address
after merge


0000_0000 strcpy: ldrb r0, [r1], #1
0000_0004 strb r0, [r2], #1
0000_0008 cmp r0, 0
0000_000C bne strcpy
0000_0010 mov pc, lr
0000_0014 strlen: ldrb r0, [r1], #1
0000_0018 add r2, #1
0000_001C cmp r0, 0
0000_0020 bne strlen
0000_0024 mov pc, lr
Placing .text section at 0x2000_0000
2000_0000 strcpy: ldrb r0, [r1], #1
2000_0004 strb r0, [r2], #1
2000_0008 cmp r0, 0
2000_000C bne strcpy
2000_0010 mov pc, lr
2000_0014 strlen: ldrb r0, [r1], #1
2000_0018 add r2, #1
2000_001C cmp r0, 0
2000_0020 bne strlen
2000_0024 mov pc, lr
Patched

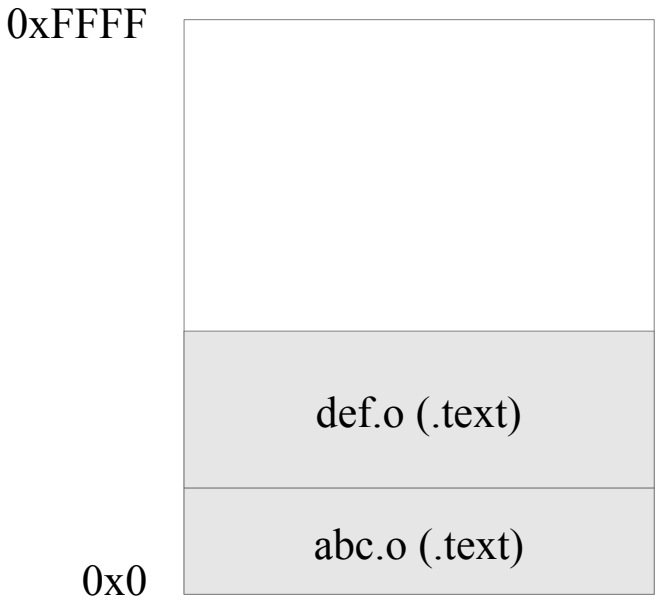
Linker Script



Can be controlled through Linker scripts.

Simple Linker Script

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
    .text : {
        abc.o (.text);
        def.o (.text);
    } > FLASH
}
```



Simple Linker Script

```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          abc.o (.text);
          def.o (.text);
     } > FLASH
}
```

Simple Linker Script

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
    .text : {
        abc.o (.text);
        def.o (.text);
    } > FLASH
}
```

Section Placement

Making it Generic

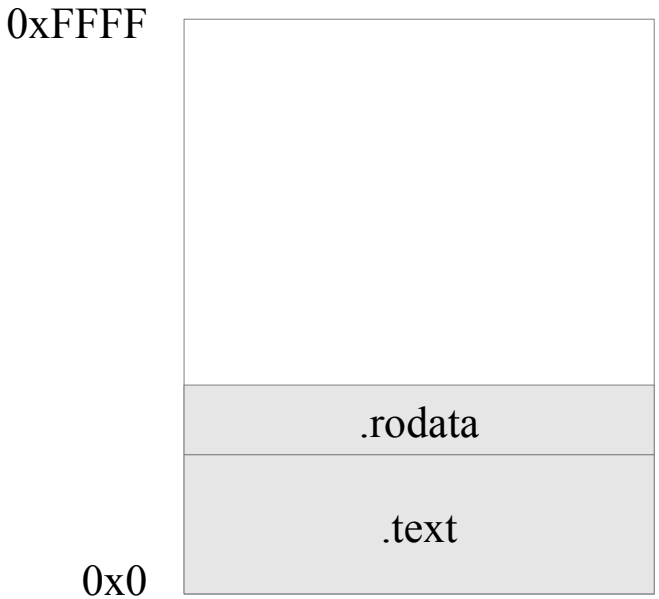
```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          * (.text);
     } > FLASH
}
```

Wildcards to represent .text form all input files

Multiple Sections

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
    .text : {
        * (.text);
    } > FLASH

    .rodata : {
        * (.rodata);
    } > FLASH
}
```

Dealing with mutiple sections



RAM is Volatile!

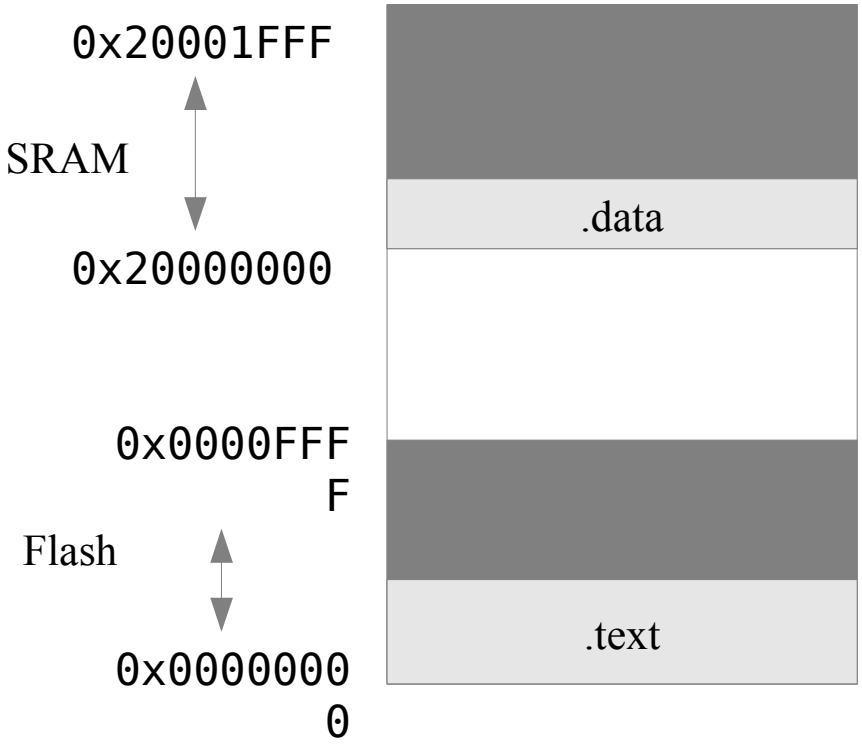
- RAM is volatile
- Data cannot be made available in RAM at power-up
- All code and data should be in Flash at power-up
- Startup code – copies data from Flash to RAM

RAM is Volatile!

- .data section should be present in Flash at power-up
- Section has two addresses
  - load address (aka LMA)
  - run-time address (aka VMA)
- So far only run-time address – actual address assigned to labels
- Load address defaults to run-time address

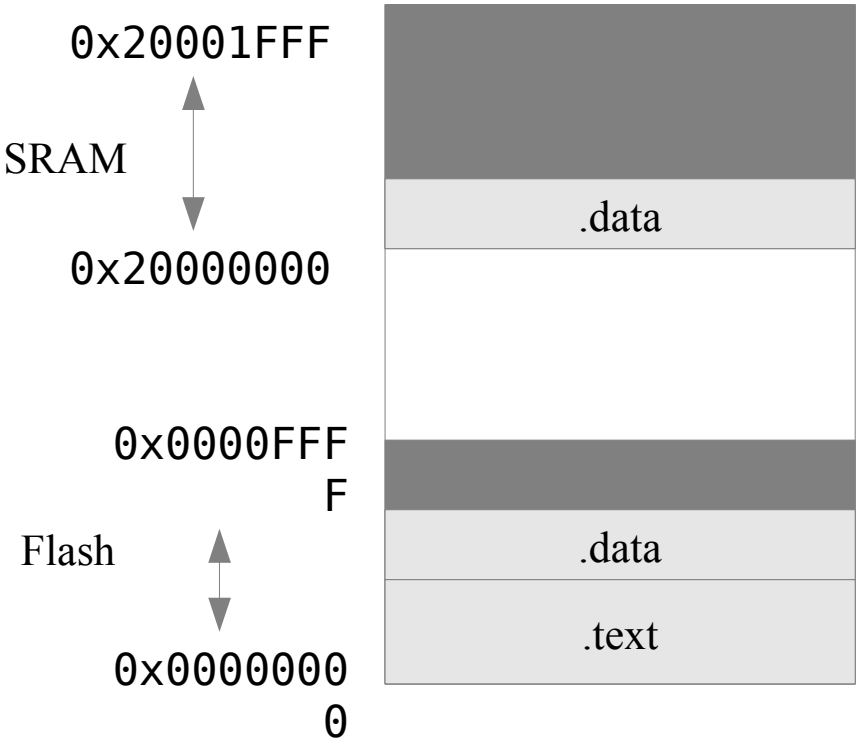
Linker Script Revisited

```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          * (.text);
     } > FLASH

     .data : {
          * (.data);
     } > SRAM
}
```



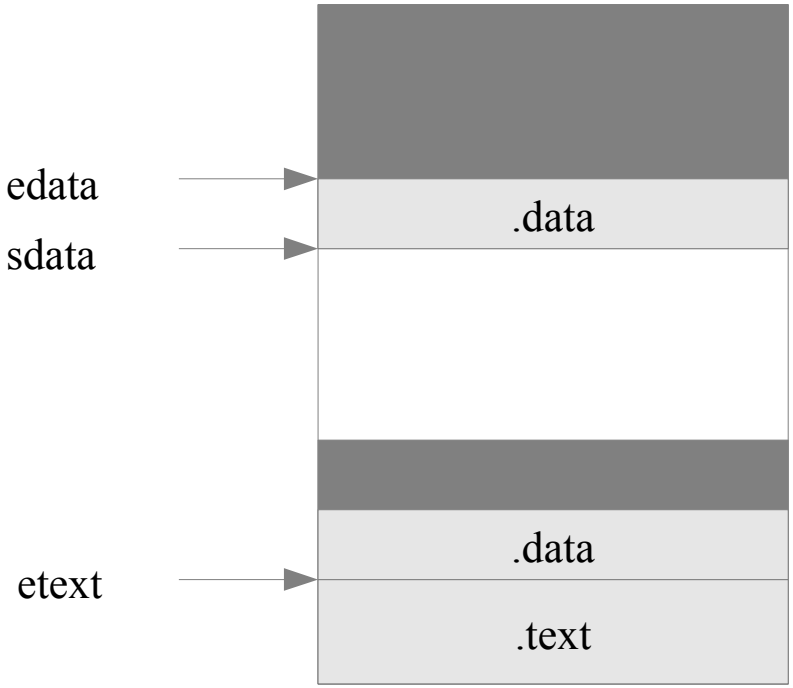
Linker Script Revisited

```
MEMORY {
     FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
     SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
     .text : {
          * (.text);
     } > FLASH

     .data : {
          * (.data);
     } > SRAM AT> FLASH
}
```



Linker Script Revisited

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x10000
    SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x2000
}

SECTIONS {
    .text : {
        * (.text);
        etext = .;
    } > FLASH

    .data : {
        sdata = .;
        * (.data);
        edata = .;
    } > SRAM AT> FLASH
}
```



Data in RAM

- Copy .data from Flash to RAM

```
start:
        ldr r0, =sdata          @ Load the address of sdata
        ldr r1, =edata          @ Load the address of edata
        ldr r2, =etext          @ Load the address of etext

copy:   ldrb r3, [r2]           @ Load the value from Flash
        strb r3, [r0]           @ Store the value in RAM

        add  r2, r2, #1         @ Increment Flash pointer
        add  r0, r0, #1         @ Increment RAM pointer

        cmp  r0, r1             @ Check if end of data
        bne  copy               @ Branch if not end of data
```

Lab-1

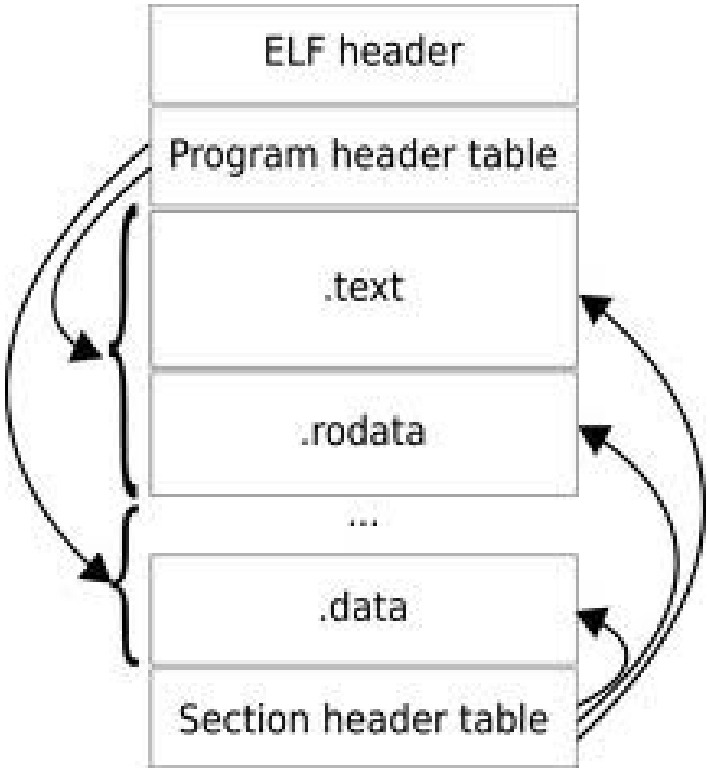
libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x/startup/gcc_ride7/startup_stm32f10x_md.s

What are the contents of typical linker script?

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

MEMORY
{
 ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

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

What does an executable image file contain?

- .text segment
  - Executable code
  - Initial reset vector
- .data segment (.rodata in ELF)
  - Static (initialized) variables
- .bss segment
  - Static (uninitialized) variables
  - Zero-filled by CRT or OS
  - From: Block Started by Symbol
- Does not contain heap or stack



- For details, see:

  /usr/include/linux/elf.h

How can the contents of an executable file be read?

- Exactly the same way as an object file!

- Recall the useful options
  - -a show all information
  - -s shows symbols
  - -S shows section headers

What does an executable file contain?

- Use readelf's –S option
- Note that the .out has fewer sections than the .o file
  - Why?

```
$ readelf -S example1.out
There are 6 section headers, starting at offset 0x8068:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 008000 000014 00  AX  0   0  4
  [ 2] .ARM.attributes   ARM_ATTRIBUTES  00000000 008014 000021 00      0   0  1
  [ 3] .shstrtab         STRTAB          00000000 008035 000031 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 008158 000130 10      5   9  4
  [ 5] .strtab           STRTAB          00000000 008288 000085 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

What are the contents of an executable's .text segment?

- Use readelf's –x option to hex dump a section
- 1st column shows memory address
- 2nd through 5th columns show data
- The initial SP and PC values are visible
- The executable opcodes are also visible

```
$ readelf -x .text example1.out

Hex dump of section '.text':
  0x00000000 | 00080020 09000000 0a200021 09180138 ... ..... .!...8
  0x00000010 | fcd1fee7                           ....
```

What are the raw contents of an executable file?

- Use hexdump
- ELF's magic number is visible
- The initial SP, PC, executable opcodes are visible

```
$ hexdump example1.out
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0028 0001 0000 0000 0000 0034 0000
0000020 8068 0000 0000 0500 0034 0020 0001 0028
0000030 0006 0003 0001 0000 8000 0000 0000 0000
0000040 0000 0000 0014 0000 0014 0000 0005 0000
0000050 8000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0000 0000 0000 0000
*
0008000 0800 2000 0009 0000 200a 2100 1809 3801
0008010 d1fc e7fe 2041 0000 6100 6165 6962 0100
0008020 0016 0000 4305 524f 4554 2d58 334d 0600
0008030 070a 094d 0002 732e 6d79 6174 0062 732e
0008040 7274 6174 0062 732e 7368 7274 6174 0062
0008050 742e 7865 0074 412e 4d52 612e 7474 6972
0008060 7562 6574 0073 0000 0000 0000 0000 0000
0008070 0000 0000 0000 0000 0000 0000 0000 0000
*
0008090 001b 0000 0001 0000 0006 0000 0000 0000
```

What purpose does an executable file serve?

- Serves as a convenient container for sections/segments
- Keeps segments segregate by type and access rights
- Serves as a program “image” for operating systems
- Allows the loader to place segments into main memory
- Can integrates symbol table and debugging information

How useful is an executable image
for most embedded systems & tools?

What does a binary program image contain?

- Basically, a binary copy of program's .text section
- Try 'hexdump –C example.bin'
- Want to change the program?
  - Try 'hexedit example.bin'
  - You can change the program (e.g. opcodes, static data, etc.)
- The initial SP, PC, executable opcodes are visible

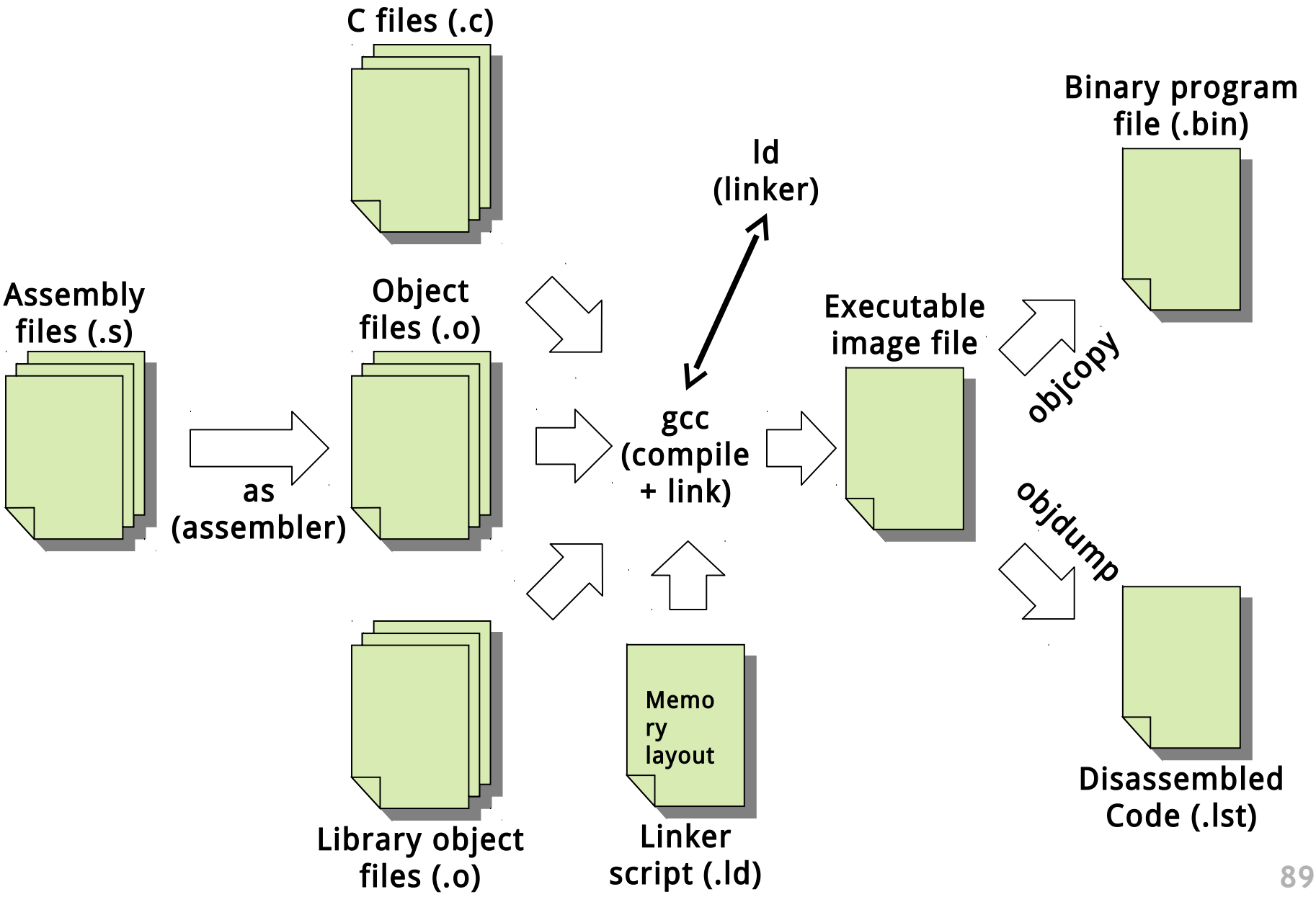
```
$ hexdump -C example.bin
00000000  00 08 00 20 09 00 00 00  0a 20 00 21 09 18 01 38  |....|
00000010  fc d1 fe e7                                       |....|
00000014

$ hexedit example.bin
00000000   00 08 00 20  09 00 00 00  0A 20 00 21  09 18 01 38  ....
00000010   FC D1 FE E7                                  ....
```

What are other, more usable formats?

- .o, .out, and .bin are all binary formats
- Many embedded tools don't use binary formats
- Two common ASCII formats
  - Intel hex (ihex)
  - Motorola S-records (srec)
- The initial SP, PC, executable opcodes are visible

```
$ arm-none-eabi-objcopy -O ihex example1.out "example1.hex "
$ cat example1.hex
:10000000000800200900000000A200021091801381A
:04001000FCD1FEE73A
:00000001FF

$ arm-none-eabi-objcopy -O srec example1.out "example1.srec "
$ cat example1.srec
S0100000657861 6D706C65312E73726563F7
S113000000080020090000000A200021091801 3816
S1070010FCD1FEE736
S9030000FC
```

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



generic hosted build

```
int factorial(int n) {
  int c;
  int result = 1;
  for (c = 1; c <= n; c++)
    result *= c;
  return result;
}
int main() {
  int i;
  int n;
  for (i = 0; i < 10; ++i) {
    n = factorial(i);
    printf("factorial(%d) = %d\n", i, n);
  }
}
```

```
$ arm-none-eabi-gcc \
  -mcpu=cortex-m3 \
  -mthumb main.c \
  -T generic-hosted.ld \
  -o factorial
$ qemu-arm -cpu cortex-m3 \
  ./factorial
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
```

sudo apt-get install qemu-system qemu-user qemu-utils

ARM Cortex-M3 Features

- Thumb-2 Instruction Set
- Bit Banding
- Integrated Peripherals
  - NVIC
  - Memory Protection Unit (MPU)
  - Debug Peripherals

System Memory Map

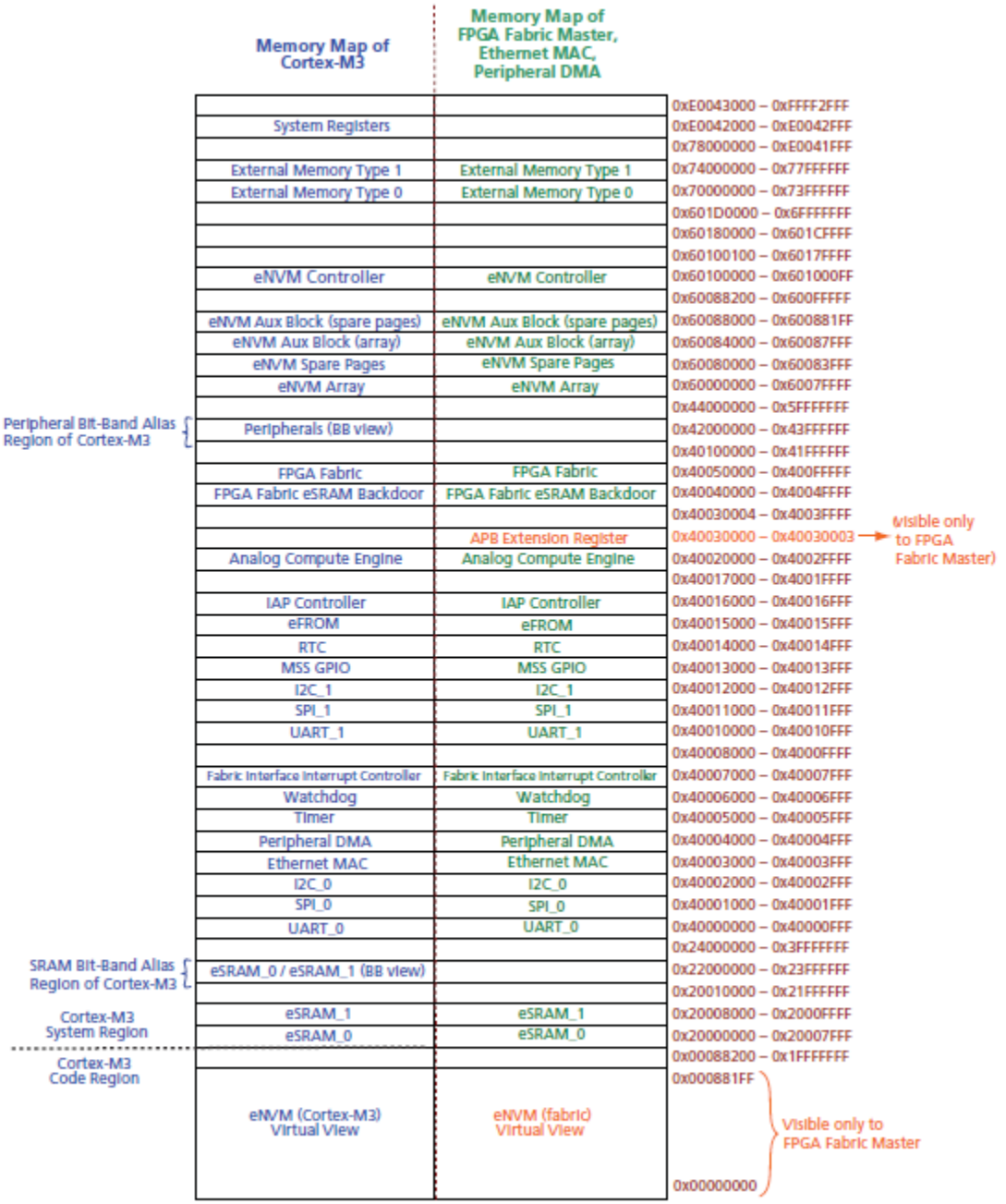


Figure 2-4 • **System Memory Map with 64 Kbytes of SRAM**

Memory-mapped I/O

- The idea is really simple
  - Instead of real memory at a given memory address, have an I/O device respond.
- 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.

Accessing memory locations from C

- Memory has an address and value
- Can equate a pointer to desired address
- Can set/get de-referenced value to change memory

```
#define  SYSREG_SOFT_RST_CR  0xE0042030

uint32_t *reg = (uint32_t *)(SYSREG_SOFT_RST_CR);

main () {
  *reg |= 0x00004000; // Reset GPIO hardware
  *reg &= ~(0x00004000);
}
```

Some useful C keywords

- const
  - Makes variable value or pointer parameter unmodifiable
  - const foo = 32;
- register
  - Tells compiler to locate variables in a CPU register if possible
  - Useless in C99
  - register int x;
- static
  - Preserve variable value after its scope ends
  - Does not go on the stack
  - static int x;
- volatile
  - Opposite of const
  - Can be changed in the background
  - volatile int I;

What happens when this "instruction" executes?

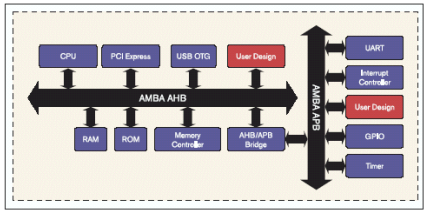
```
#include <stdio.h>
#include <inttypes.h>

#define REG_FOO 0x40000140

main () {
  uint32_t *reg = (uint32_t *)(REG_FOO);
  *reg += 3;

  printf( "0x%x\n" , *reg);
}
```

“*reg += 3” is turned into a ld, add, str sequence

- Load instruction
  - A bus read operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a read operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The target drives the contents of “reg” onto the data lines
  - The contents of “reg” is loaded into a CPU register (e.g. r0)
- Add instruction
  - An immediate add (e.g. add r0, #3) adds three to this value
- Store instruction
  - A bus write operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a write operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The CPU drives the contents of “r0” onto the data lines
  - The target stores the data value into address “reg”

Details of the bus “handshaking” depend on the particular memory/peripherals involved

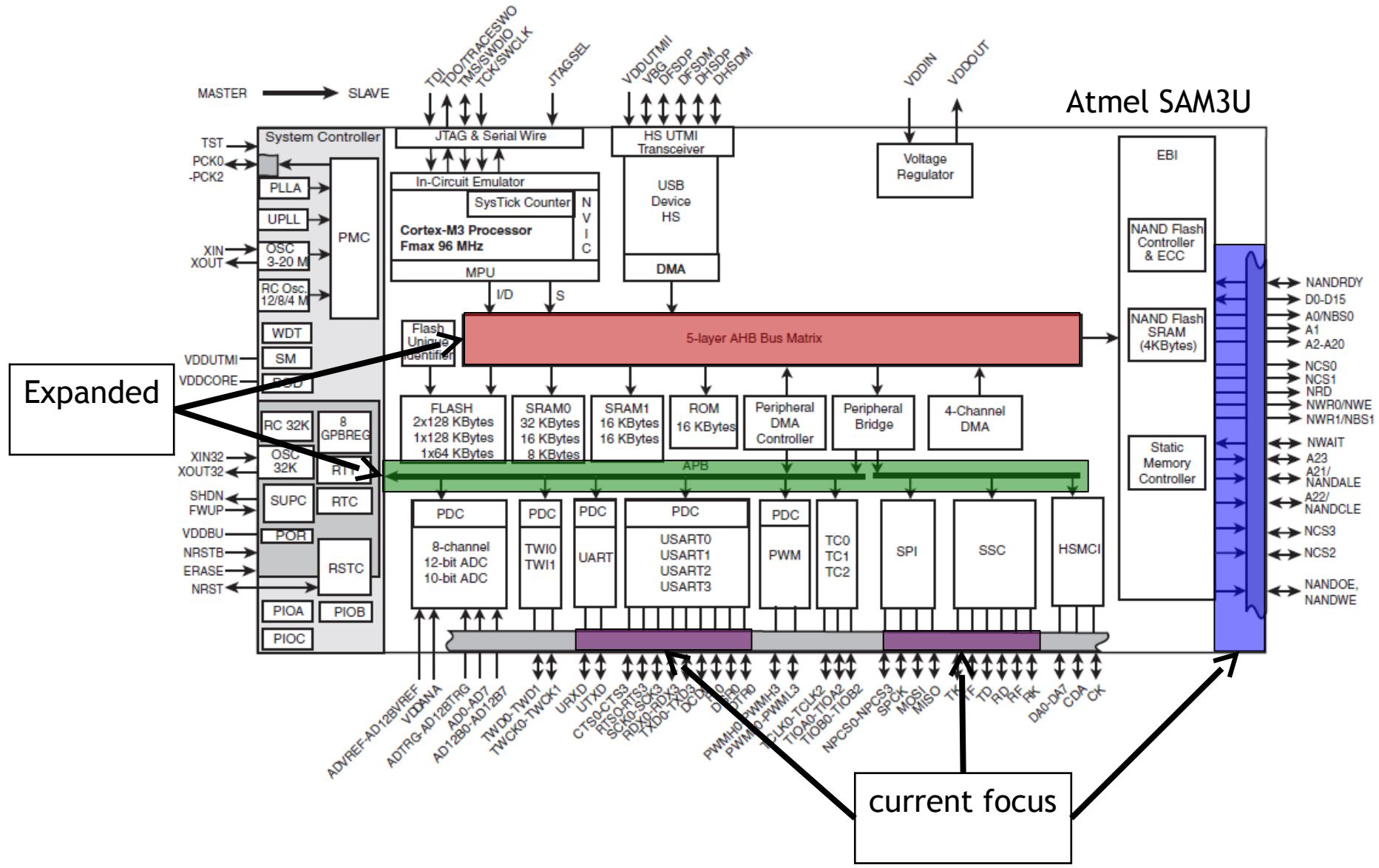
- SoC memory/peripherals
  - AMBA AHB/APB



- NAND Flash
  - Open NAND Flash Interface (ONFI)

- DDR SDRAM
  - JEDEC JESD79, JESD79-2F, etc.

Modern embedded systems have multiple busses

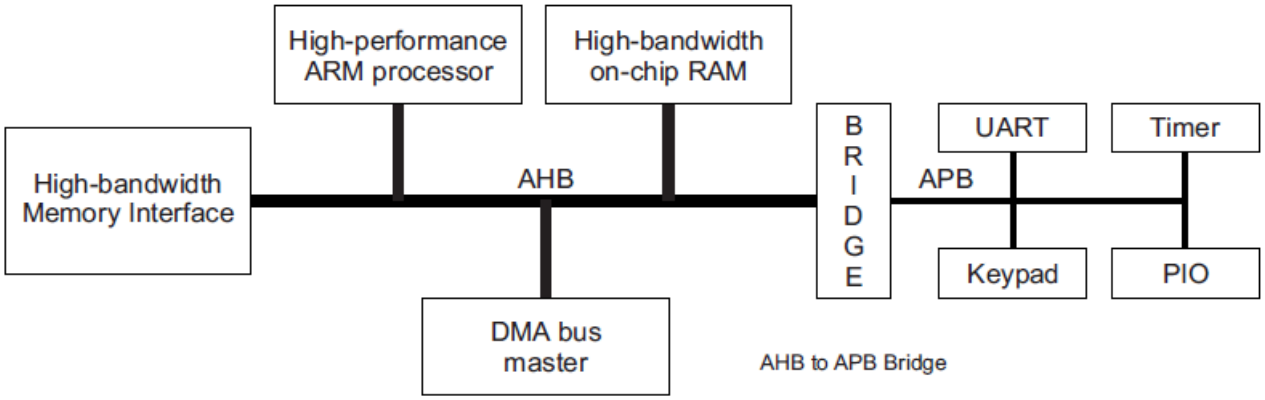


Why have so many busses?

- Many designs considerations
  - Master vs Slave
  - Internal vs External
  - Bridged vs Flat
  - Memory vs Peripheral
  - Synchronous vs Asynchronous
  - High-speed vs low-speed
  - Serial vs Parallel
  - Single master vs multi master
  - Single layer vs multi layer
  - Multiplexed A/D vs demultiplexed A/D

- Discussion: what are some of the tradeoffs?

Advanced Microcontroller Bus Architecture (AMBA)
- Advanced High-performance Bus (AHB)
- Advanced Peripheral Bus (APB)



AHB

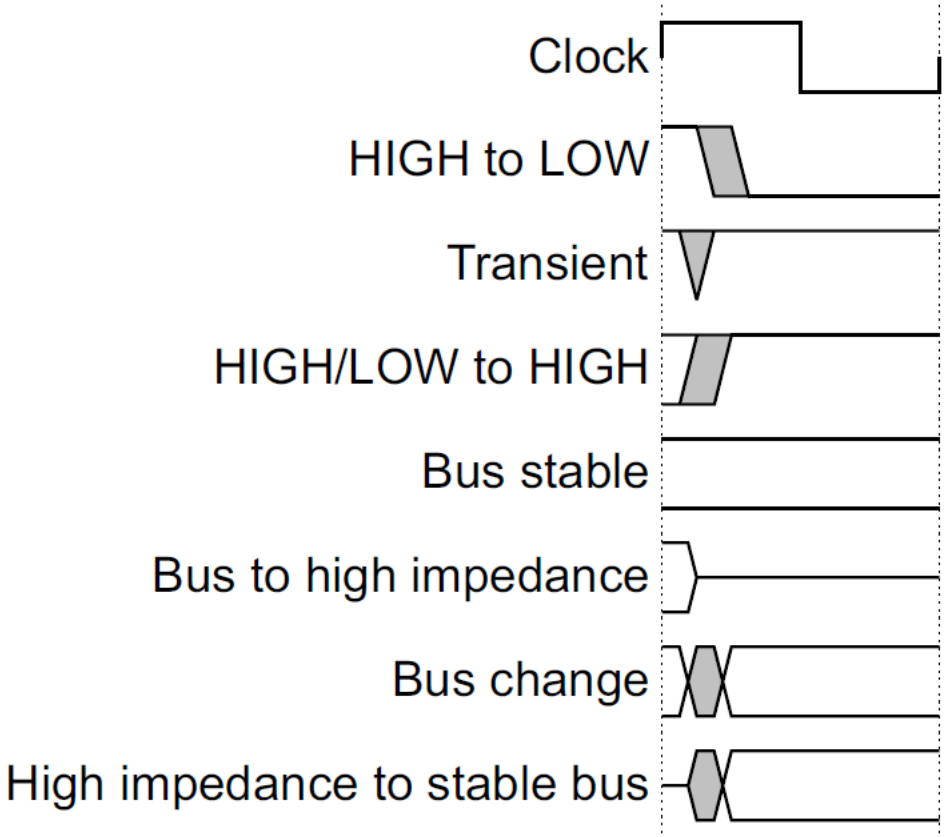
- High performance
- Pipelined operation
- Burst transfers
- Multiple bus masters
- Split transactions

APB

- Low power
- Latched address/control
- Simple interface
- Suitable of many peripherals

Key to timing diagram conventions

- Timing diagrams
  - Clock
  - Stable values
  - Transitions
  - High-impedance
- Signal conventions
  - Lower case ‘n’ denote active low (e.g. RESETn)
  - Prefix ‘H’ denotes AHB
  - Prefix ‘P’ denotes APB



Basic read and write transfers with no wait states

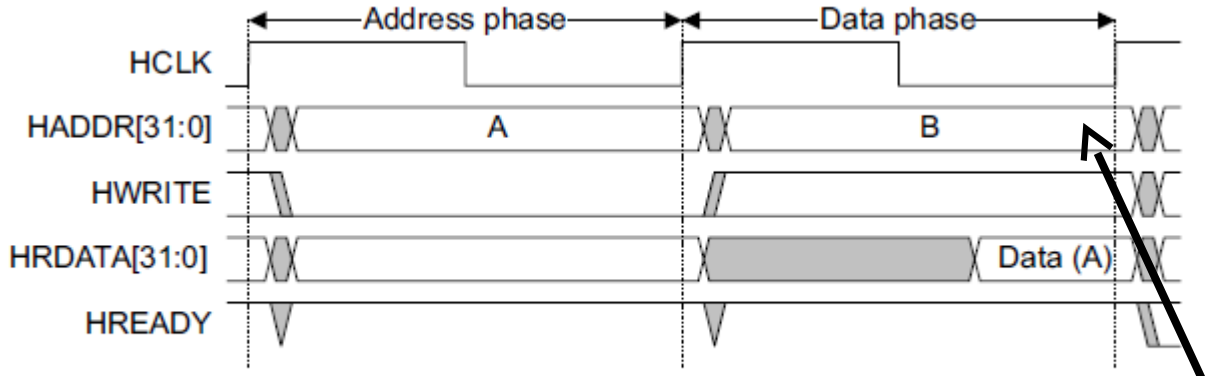


Figure 3-1 Read transfer

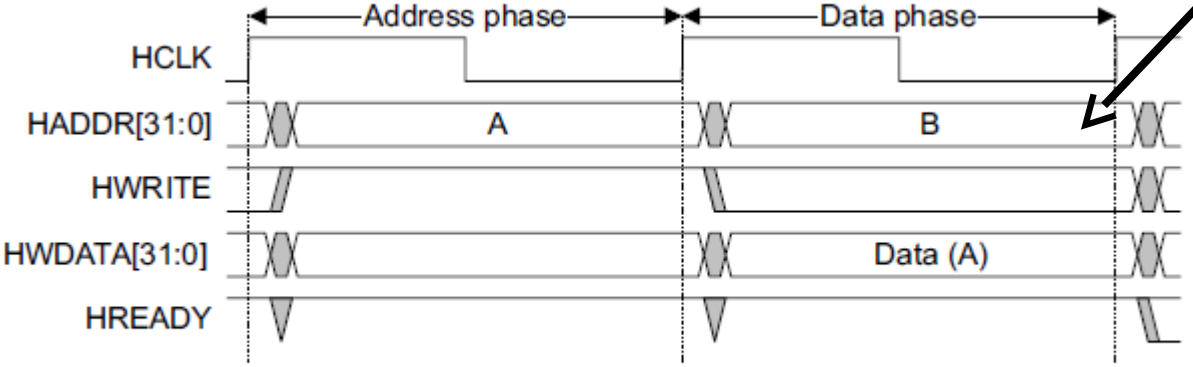
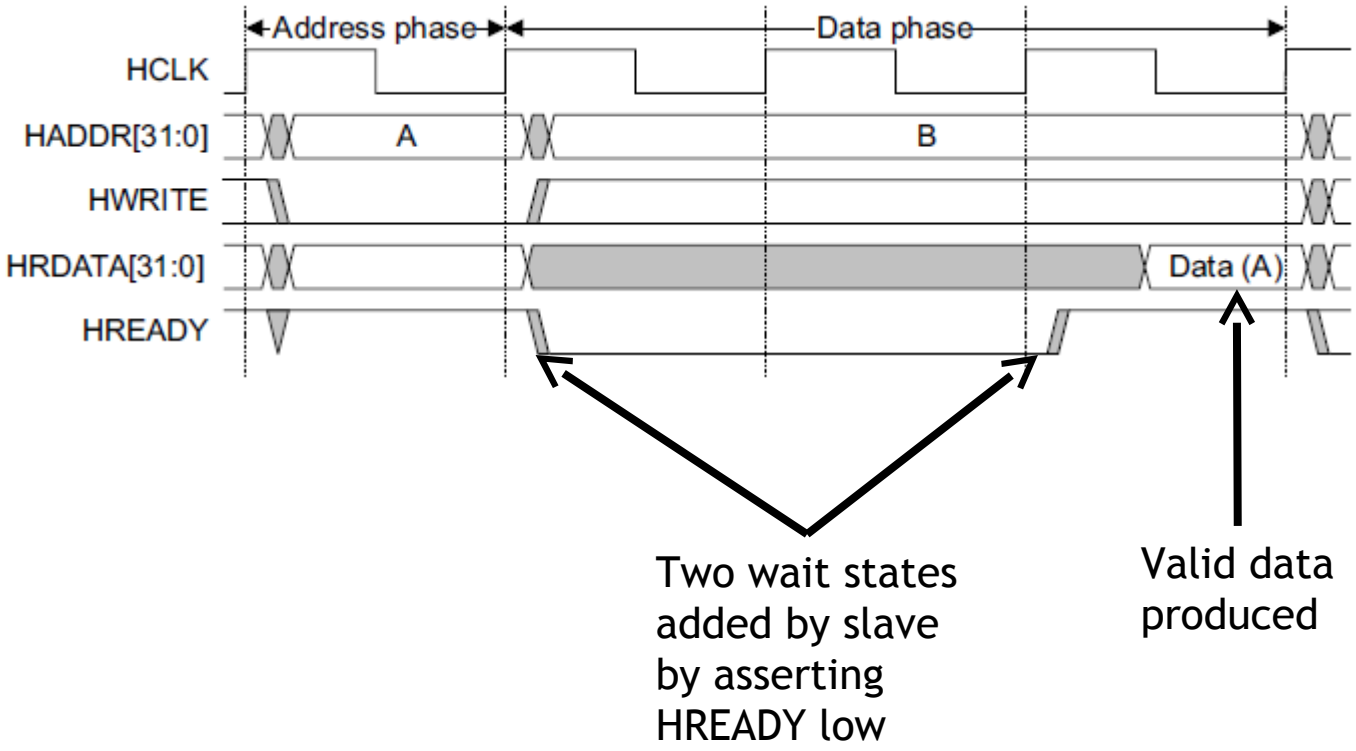


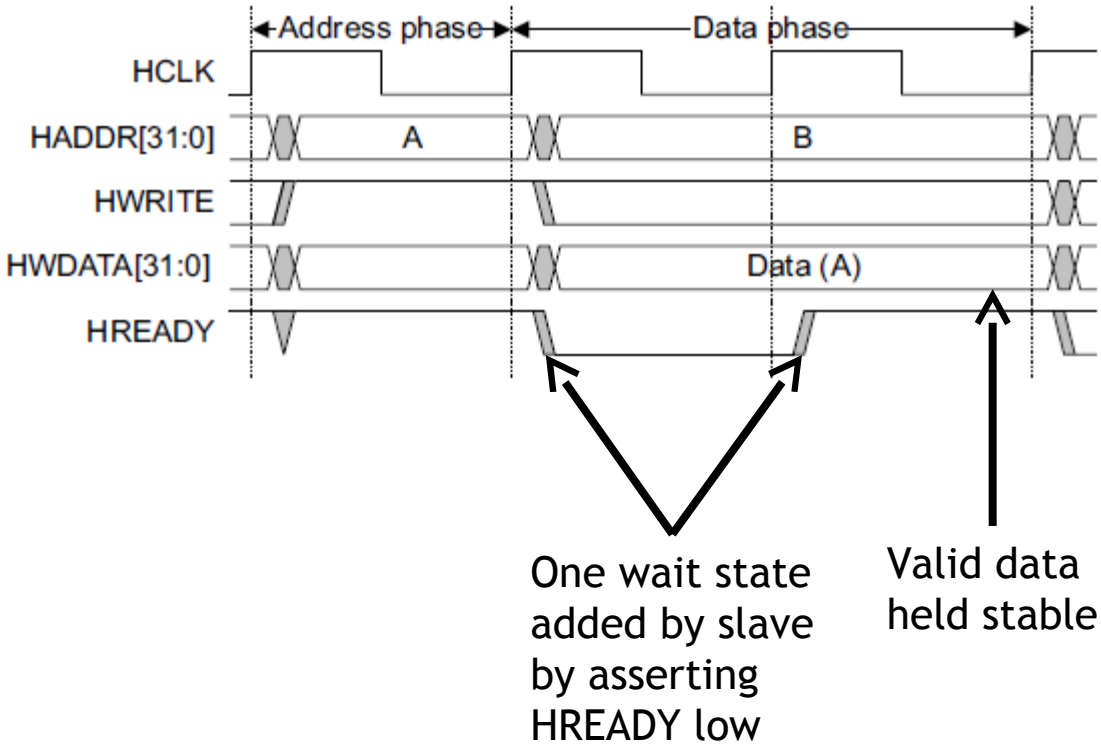
Figure 3-2 Write transfer

Pipelined Address & Data Transfer

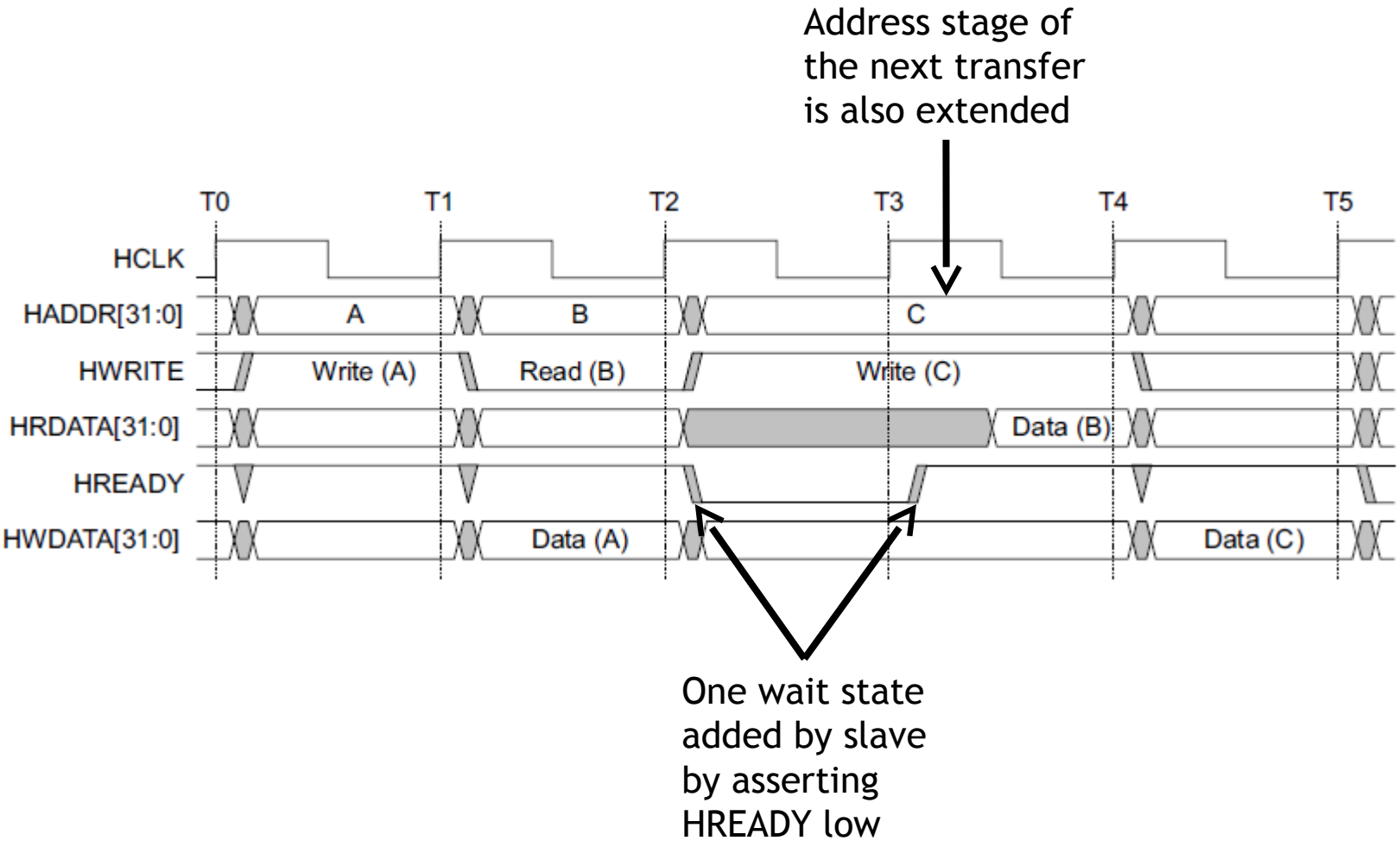
Read transfer with two wait states



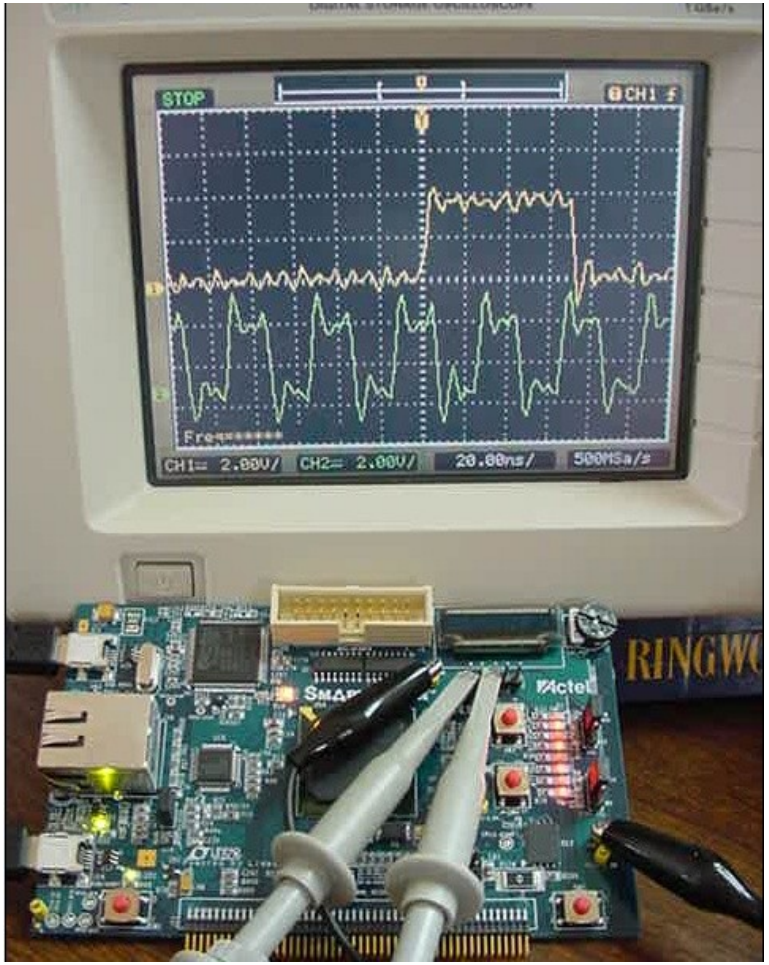
Write transfer with one wait state



Wait states extend the address phase of next transfer



memory-mapped peripheral



I/O Data Transfer

Two key questions to determine how data is transferred to/from a non-trivial I/O device:

1. How does the CPU know when data is available?
   a. Polling
   b. Interrupts

2. How is data transferred into and out of the device?
   a. Programmed I/O
   b. Direct Memory Access (DMA)

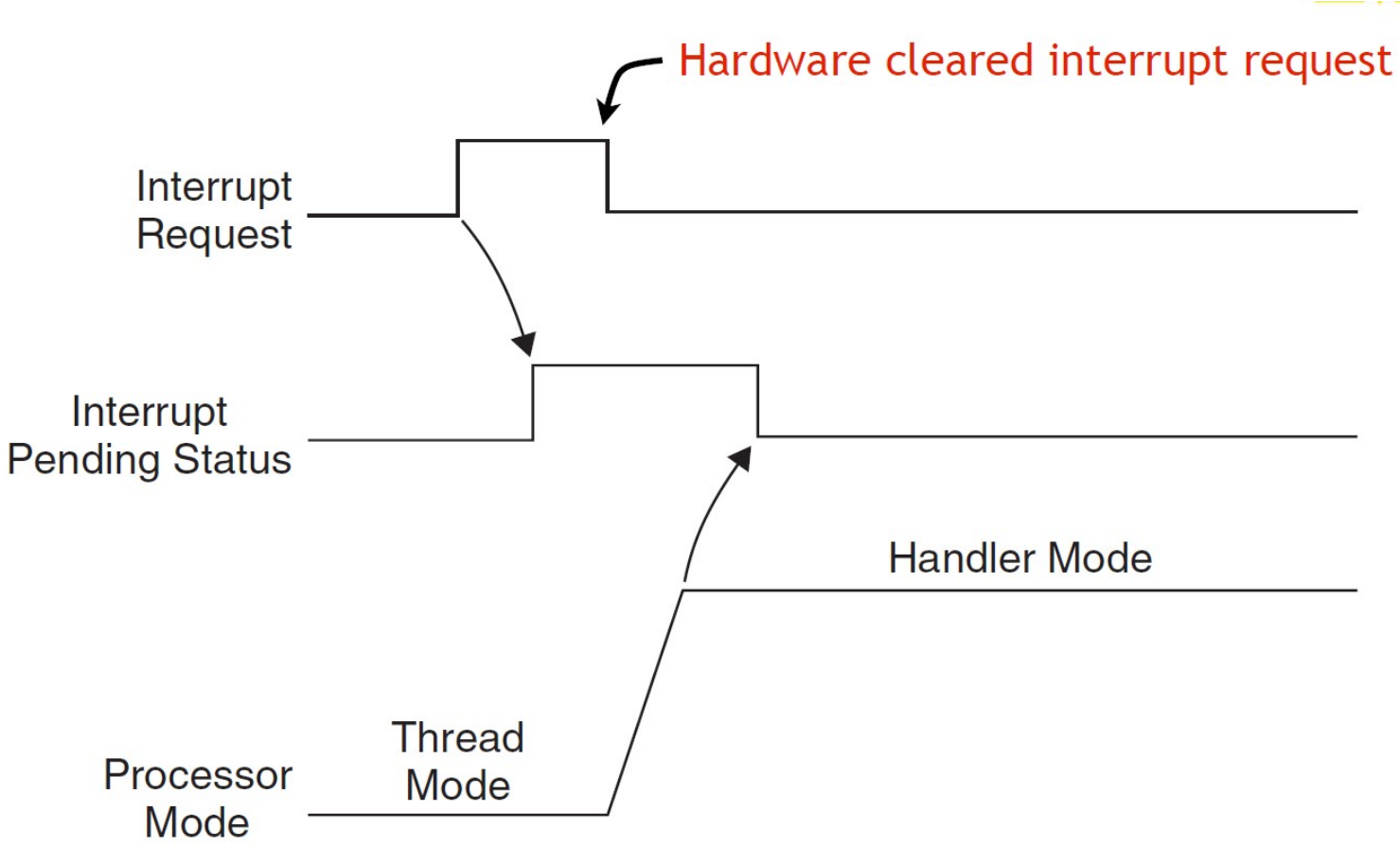
Two basic types of interrupts (1/2)

- Those caused by an instruction
  - Examples:
    - TLB miss
    - Illegal/unimplemented instruction
    - div by 0
  - Names:
    - Trap, exception

Two basic types of interrupts (2/2)

- Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error
- Names:
  - interrupt, external interrupt

Pending interrupts

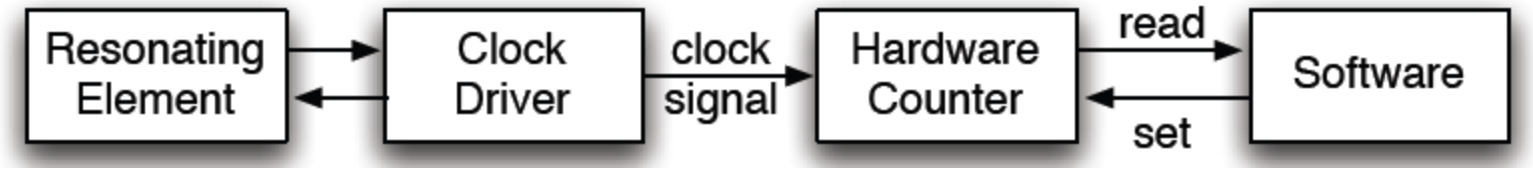


The normal case. Once Interrupt request is seen, processor puts it in "pending" state even if hardware drops the request.
IPS is cleared by the hardware once we jump to the ISR.

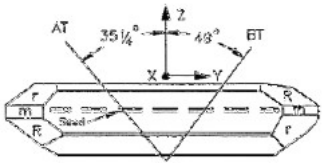
Fine grain motion control

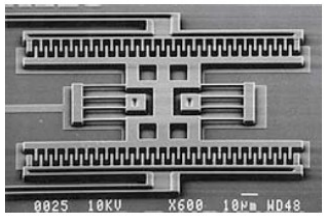
http://www.youtube.com/watch?v=SOESSCXGhFo

Clock generation and use



- Resonating element/Driver:
  - Quartz crystal can be made to resonate due to Piezoelectric effect.
    - Resonate frequency depends on length, thickness, and angle of cut.
    - Issues: Very stable (<100ppm) but not all frequencies possible.
  - MEMS Resonator
    - Arbitrary frequency, potentially cheap, susceptible to temperature variations.
  - Others:
    - Inverter Ring, LC/RC circuits, Atomic clock, and many more.





Appendix

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

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