UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 12: Assembly Process, Macros, Memory Maps, Linking/Loading

Taylor Johnson

Announcements and Outline

- Quiz 4 upcoming
  - Will help review for midterm next week
- Homework 4 due 10/7
- Midterm 10/9
  - Chapter 1, 2 (ARM), Appendices A1-A6, Appendices B1-B2 (ARM)

- Review: Control Flow, Stack, Recursion
  - Basic function calls
  - Intro to Recursion and the Stack
- Macros, pseudoinstructions, assembler directives
- Assembly process

Review: The Stack

- Last-in, first-out (LIFO) data structure
  - Last data put in comes out first
  - Common analogy: like a quarter / coin holder in your car, the last coin put in comes out first
- Stack pointer (SP) register: points to current address of stack (i.e., the last thing in)
  - **YOU** must initialize it! Typically use address 0x100000
  - `mov sp, #0x100000`
- Stack instructions
  - `PUSH {r0}` means:
    - `SUB sp, sp, #4`
    - `STR r0, [sp]`
  - `POP {r0}` means:
    - `LDR r0, [sp]`
    - `ADD sp, sp, #4`
  - Can use lists of registers, e.g., `PUSH {r0,r1}` is:

```
SUB sp, sp, #8
STR r0, [sp]
STR r1, [sp,#4]
```

Review: Summary of Caller and Callee Steps

- Caller steps:
  - Step 1: Put arguments in the registers r0, r1, r2, r3.
  - Step 2: Branch to the function, using the bl instruction.
  - Step 3: After the function has returned, recover the return value (if any), and use it.
- Callee (called function) steps:
  - Step 1 (preamble): Allocate memory on the stack, and save register rl, and other registers that the function modifies, to the stack.
  - Step 2: Do the main body of the function.
  - Step 3 (wrap-up):
    - Store the return value (if any) on r0, second return value (if any) on r1.
    - Restore, from the stack, the original values of all registers that the function modified, as well as the value of register lr.
    - Deallocate memory on the stack (increment sp).
    - Branch to the return address using instruction bx.

Review: State Preservation Across Procedure Calls

Preserved	Not preserved
Variable registers: r4-r11	Argument registers: r0-r3
Stack pointer register: sp	Intra-procedure-call scatch register: r12
Link register: lr	Stack below the stack pointer
Stack above the stack pointer	

Review: Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int fact(int n)
{
  if (n== 0) return 1;
  return n * fact(n - 1);
}
```

- How do we write function factorial in assembly?

Review: Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int fact(int n)
{
  if (n== 0) return 1;
  return n * fact(n - 1);
}
```

- How do we write function factorial in assembly?

```
@ factorial main body
mov r4, r0
cmp r4, #0
moveq r0, #1
beq fact_exit

sub r0, r4, #1
bl fact
mov r5, r0
mul r0, r5, r4
```

Review: Recursive Function Example: Factorial

```
    @ factorial preamble
fact:     ???

    @ factorial body
    mov r4, r0
    cmp r4, #0
    moveq r0, #1
    beq fact_exit

    sub r0, r4, #1
    bl fact
    mov r5, r0
    mul r0, r5, r4
```

```
 @ factorial wrap-up
fact_exit:          ???
```

Review: Recursive Function Example: Factorial

```
      @ factorial preamble
fact: sub sp, sp, #12
      str lr, [sp, #0]
      str r4, [sp, #4]
      str r5, [sp, #8]

      @ factorial body
      mov r4, r0
      cmp r4, #0
      moveq r0, #1
      beq fact_exit

      sub r0, r4, #1
      bl fact
      mov r5, r0
      mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
   ldr lr, [sp, #0]
   ldr r4, [sp, #4]
   ldr r5, [sp, #8]
   add sp, sp, #12
   bx lr
```

Review: Recursive Function Example: Factorial

```
    @ factorial preamble
fact: push {r4,r5,lr}

    @ factorial body
    mov r4, r0
    cmp r4, #0
    moveq r0, #1
    beq fact_exit

    sub r0, r4, #1
    bl fact
    mov r5, r0
    mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
    pop {r4,r5,lr}
    bx lr
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x5      5
r1             0x183    387
r2             0x100    256
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0xfff4   0xfff4
lr             0x1000c  65548
pc             0x10014  0x10014 <fact+4>
cpsr           0x600001d3       1610613203
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x4      4
r1             0x183    387
r2             0x100    256
r3             0x0      0
r4             0x5      5
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0xffe8   0xffe8
lr             0x1002c  65580
pc             0x10014  0x10014 <fact+4>
cpsr           0x200001d3     536871379
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0      0x3     3
r1      0x183   387
r2      0x100   256
r3      0x0     0
r4      0x4     4
r5      0x0     0
r6      0x0     0
r7      0x0     0
r8      0x0     0
r9      0x0     0
r10     0x0     0
r11     0x0     0
r12     0x0     0
sp      0xffdc  0xffdc
lr      0x1002c 65580
pc      0x10014 0x10014 <fact+4>
cpsr    0x200001d3      536871379
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0      0x2     2
r1      0x183   387
r2      0x100   256
r3      0x0     0
r4      0x3     3
r5      0x0     0
r6      0x0     0
r7      0x0     0
r8      0x0     0
r9      0x0     0
r10     0x0     0
r11     0x0     0
r12     0x0     0
sp      0xffd0  0xffd0
lr      0x1002c 65580
pc      0x10014 0x10014 <fact+4>
cpsr    0x200001d3      536871379
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x1      1
r1             0x183    387
r2             0x100    256
r3             0x0      0
r4             0x2      2
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0xffc4   0xffc4
lr             0x1002c  65580
pc             0x10014  0x10014 <fact+4>
cpsr           0x200001d3     536871379
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x0      0
r1             0x183    387
r2             0x100    256
r3             0x0      0
r4             0x1      1
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0xffb8   0xffb8
lr             0x1002c  65580
pc             0x10014  0x10014 <fact+4>
cpsr           0x200001d3      536871379
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x78      120
r1             0x183     387
r2             0x100     256
r3             0x0       0
r4             0x0       0
r5             0x0       0
r6             0x0       0
r7             0x0       0
r8             0x0       0
r9             0x0       0
r10            0x0       0
r11            0x0       0
r12            0x0       0
sp             0x10000   0x10000 <_start>
lr             0x1000c   65548
pc             0x1000c   0x1000c <iloop>
cpsr           0x600001d3     1610613203
```

Review: Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Stack after final return:

0xff90:    0      0      0      0
0xffa0:    0      0      0      0
0xffb0:    0      0      1      0
0xffc0:    65580  2      0      65580
0xffd0:    3      0      65580  4
0xffe0:    0      65580  5      0
0xfff0:    65580  0      0      65548
0x10000
```

UNIVERSITY OF TEXAS ARLINGTON

Review: Convert Iterative Factorial to Assembly

```
int factorial(int n) {
    int f = 1;
    while (n > 0) {
        f *= n;
        n--;
    }
    return f;
}
```

```
factorial:  ???
```

Array Example

```
.globl _start
_start: mov        r1,#0 @ r1 := 0
        ldr    r0,=arrayPtr       @ r0 := arrayPtr
        ldr    r3,=arrayEnd       @ r3 := arrayEnd
        ldrb   r4,[r3,#0]  @ r4 := MEM[R3 + 0]
loop:   ldrb   r2,[r0,#0]  @ r3 := MEM[r0]
        cmp    r2,r4       @ r0 == 0xFF ?
        beq    done        @ branch if done
        add    r1,r1,r2    @ r1 := r1 + r2
        add    r0,r0,#1    @ r0 := r0 + #1
        b      loop        @ pc = loop (address)
done:   strb   r1,[r2]     @ MEM[r2] := r1
iloop:         b     iloop @ infinite loop
```

```
arrayPtr:
        .byte 2
        .byte 3
        .byte 5
        .byte 7
        .byte 11
        .byte 13
        .byte 17
        .byte 19
        .byte 23
        .byte 29
        .byte 31
        .byte 37
        .byte 41
        .byte 43
        .byte 47
arrayEnd:
        .byte 0xFF
```

Macros

- Another assembler directive
  - Like .byte, .word, .asciz, that we've seen a little of before
- Way to refer to commonly used or repeated code
- Similar to an assembly procedure or function, ***but expanded (evaluated) at assembly time***, not run time
- Similar to #define in C, which is replaced by compiler at compile time
- Macro call: use of macro as an instruction
- Macro expansion: replacement of macro body by the corresponding instructions

Macros vs. Procedures

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Macro Example

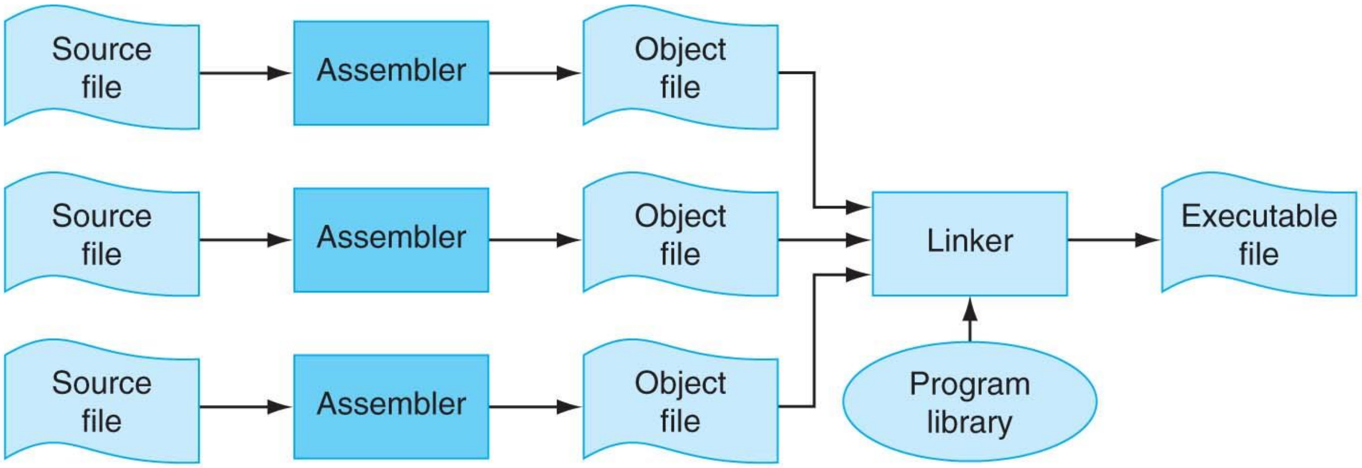
```
.globl _start
_start: .macro addVals adA, adB
        ldrb    r2,[\adA]       @ r2 := MEM[adA]
        ldrb    r3,[\adB]       @ r3 := MEM[adB]
        sub     r5,r2,r3        @ r5 := r2 - r3 = A - B
        strb    r5,[\adA]       @ MEM[adA] = r5
        ldrb    r2,[\adA]       @ r2 := MEM[adA]
        add     \adA,\adA,#1    @ r0 := r0 + 1
        add     \adB,\adB,#1    @ r1 := r1 + 1
        .endm                   @ end macro definition
init:   ldr     r0,=A           @ r0 := A (address)
        ldr     r1,=B           @ r1 := B (address)
        ldr     r4,=A_end       @ r4 := A_end (address)
        addVals       r0,r1     @ call macro
done:   b       done            @ infinite loop
```

```
A:      .byte 9, 8, 7, 6
A_end:      .byte 0
B:      .byte 1, 1, 1, 1
B_end:      .byte 0
```

Assembly Process

- Insuffiency of one pass
  - Suppose we have labels (symbols).
  - How do we calculate the addresses of labels later in the program?
  - Example:
    - `ADDR: 0x1000          b` **`done`**
    - … `// Other instructions and data`
    - `ADDR: 0x????` **`done:`** `add r1, r2, r0`
    - …
    - How to compute address of label **`done`**?
- Two-Pass Assemblers
  - First Pass: iterate over instructions, build a symbol table, opcode table, expand macros, etc.
  - Second Pass: iterate over instructions, printing equivalent machine language, plugging in values for labels using symbol table

Assembly Process



The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Linking and Loading

- Linking: combining multiple program modules (pieces of code) into executable program
  - Examples: using our _tests files to load inputs to your programs, calling library functions like printf, etc.
- Loading: getting executable running on machine
  - Examples: calling QEMU with our binary
- Static linking
  - Combine multiple object files into single binary
- Dynamic linking
  - Load library shared code at runtime
  - Not talking about this: operating system concept
  - Examples: Windows DLLs

Makefile Example

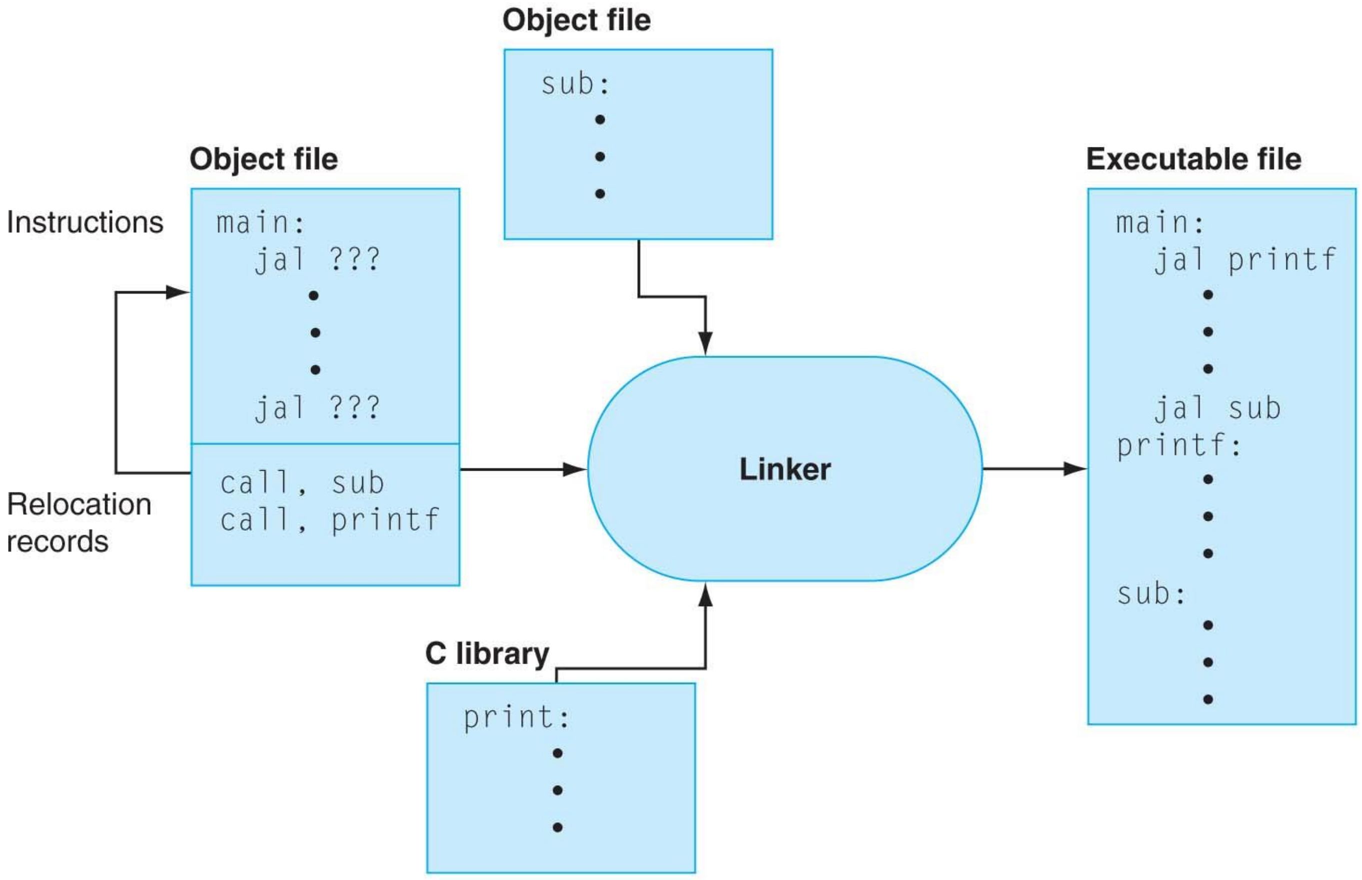
```
CROSS_COMPILE ?= arm-none-eabi
AOPS = --warn --fatal-warnings -g
example.bin : example.s example_tests.s example_memmap
	$(CROSS_COMPILE)-as $(AOPS) example.s -o example.o
	$(CROSS_COMPILE)-as $(AOPS) example_tests.s –o example_tests.o
	$(CROSS_COMPILE)-ld example.o example_tests.o -T
		example_memmap -o example.elf
	$(CROSS_COMPILE)-objdump -D example.elf > example.list
	$(CROSS_COMPILE)-objcopy example.elf -O binary example.bin
```

Linker

- `ld`
  - For us: `arm-none-eabi-ld`
  - GNU ARM linker
- Operations
  - Copy code from each input file into resulting binary
  - Resolve references between files
  - Relocate symbols to use absolute memory addresses instead of relatives
  - Binary format

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

Linker Process



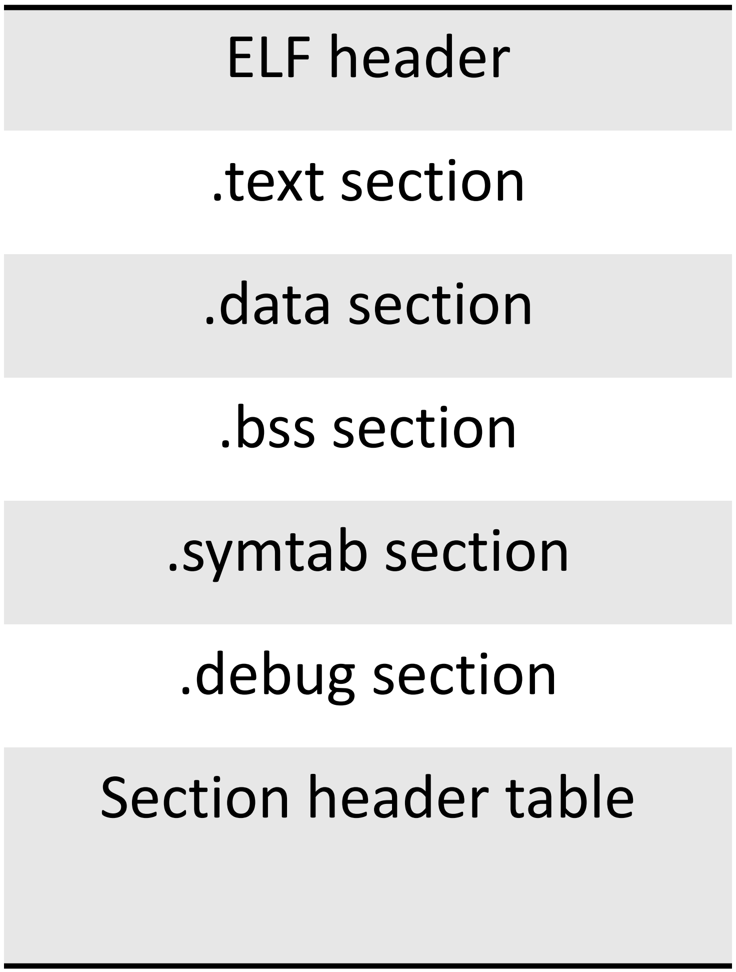
Executable and Linkable Format (ELF)

- Formerly: extensible linking format
- Standard *nix binary file format
- Unified format
  - Objects (*.o)
  - Shared objects (*.so)
  - Executables
- Header: type (.o, .os, executable), machine (ARM), byte ordering (Endianness)
- .text section: code
- .data section: initialized globals

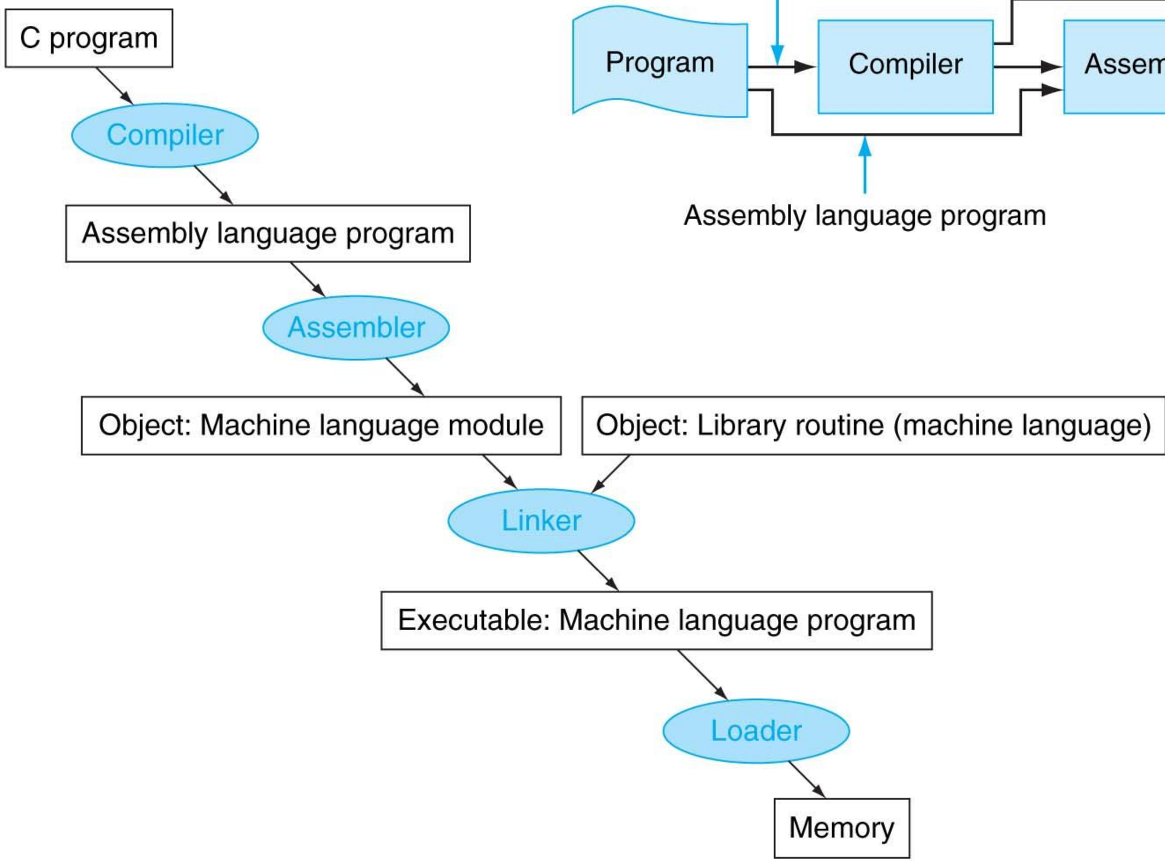
ELF header
.text section
.data section
.bss section
.symtab section
.debug section
Section header table

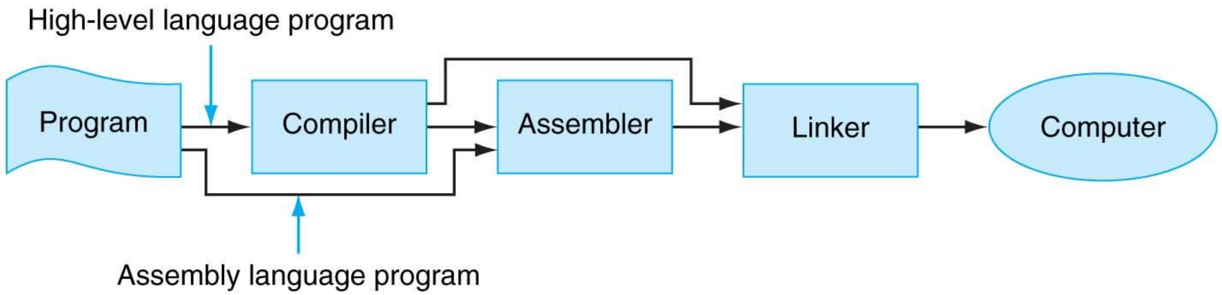
Executable and Linkable Format (ELF)

- .bss: uninitialized globals
- .symtab: symbol table
  - Label names
- .debug section: extra info for using gdb
- Section header: sizes of sections and offsets

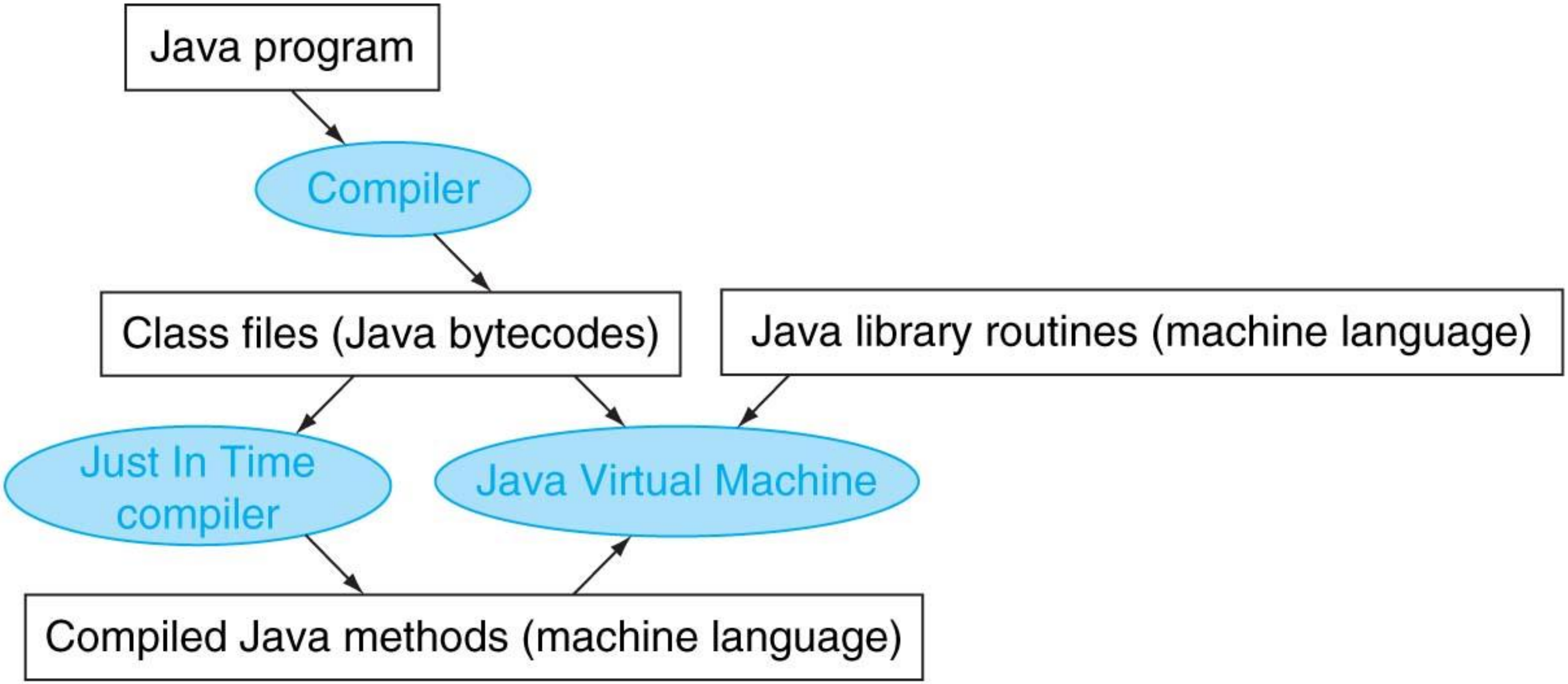


Compilation and Assembly Process (C + Assembly)





Other Languages



Memory Maps and Organization

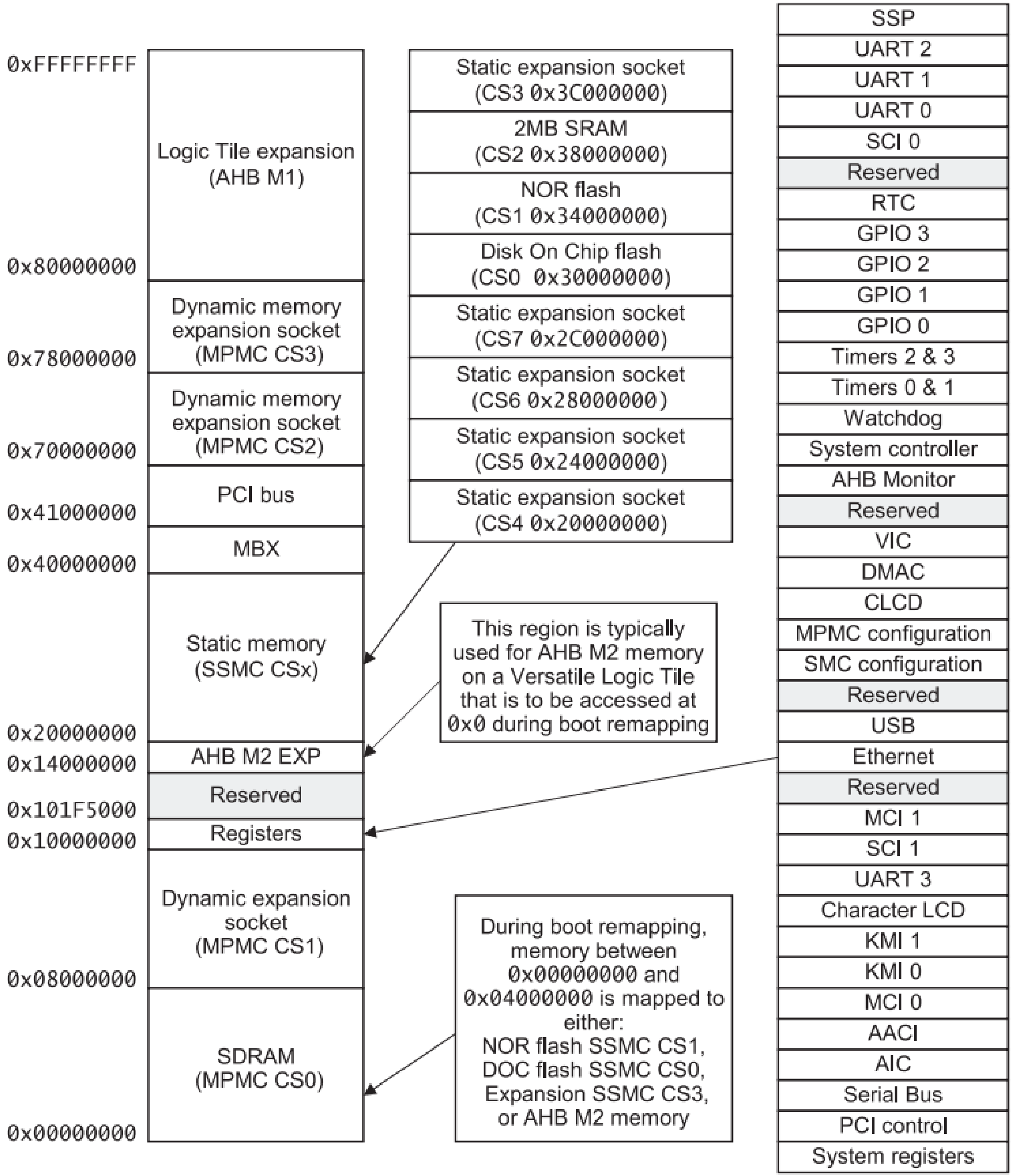
- Map: “a diagrammatic representation of an area of land or sea showing physical features, cities, roads, etc.”
- Memory map: diagrammatic representation of an area of memory showing addresses, etc.

Memory-Mapped I/O Example

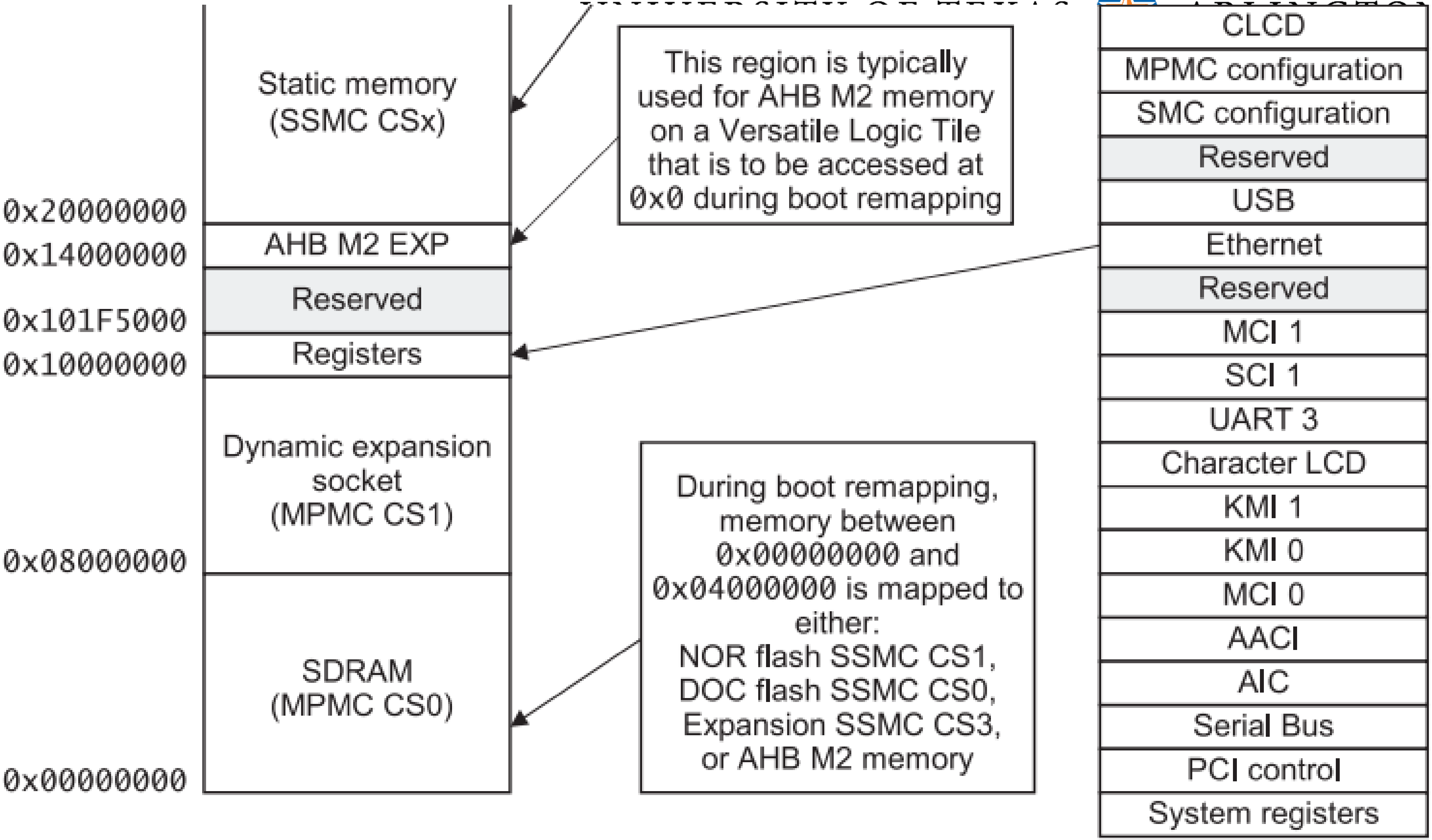
- Some of our original examples displayed output to console by writing to a special memory address

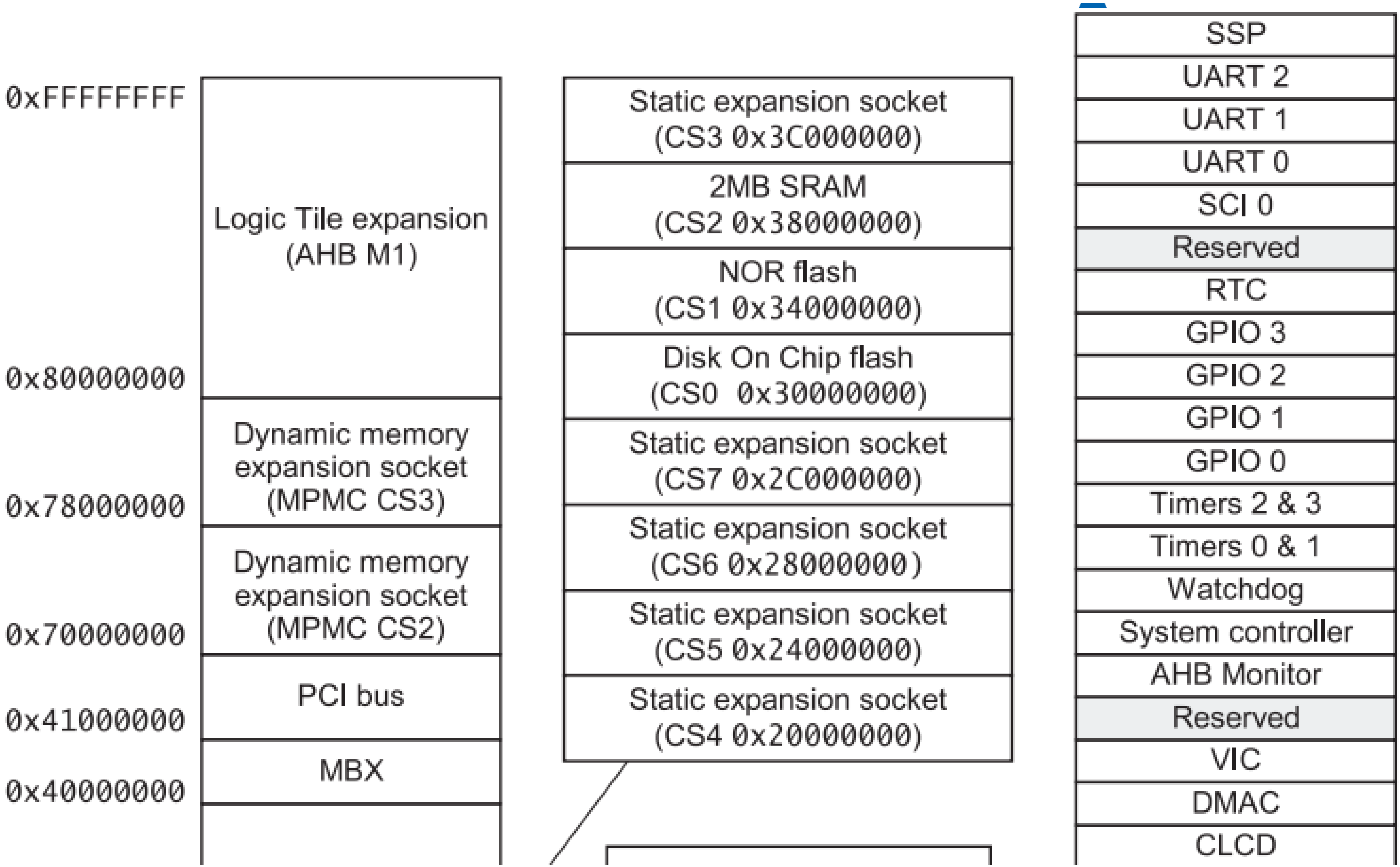
```
.equ        ADDR_UART0, 0x101f1000
ldr         r0,=ADDR_UART0 @ r0 := 0x 101f 1000
mov         r2,#0x0D               @ R2 := 0x0D (return \r)
str         r2,[r0]                @ MEM[r0] := r2
```

- How does this work?
  - Registers on peripheral devices (keyboards, monitors, network controllers, etc.) are addressable in same address space as main memory



0xFFFFFFFF
Logic Tile expansion (AHB M1)
0x80000000
Dynamic memory expansion socket (MPMC CS3)
0x78000000
Dynamic memory expansion socket (MPMC CS2)
0x70000000
PCI bus
0x41000000
MBX
0x40000000
Static memory (SSMC CSx)
0x20000000
AHB M2 EXP
0x14000000
Reserved
0x101F5000
Registers
0x10000000
Dynamic expansion socket (MPMC CS1)
0x08000000
SDRAM (MPMC CS0)
0x00000000
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers


Static memory
(SSMC CSx)
0x20000000
AHB M2 EXP
0x14000000
Reserved
0x101F5000
Registers
0x10000000
Dynamic expansion
socket
(MPMC CS1)
0x08000000
SDRAM
(MPMC CS0)
0x00000000
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either:
NOR flash SSMC CS1,
DOC flash SSMC CS0,
Expansion SSMC CS3,
or AHB M2 memory
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers


0xFFFFFFFF
0x80000000
0x78000000
0x70000000
0x41000000
0x40000000
Logic Tile expansion (AHB M1)
Dynamic memory expansion socket (MPMC CS3)
Dynamic memory expansion socket (MPMC CS2)
PCI bus
MBX
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD

Address from Memory-Map in Manual

Programmer's Reference

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
UART 0 Interface	Dev. chip	PIC 12	0x101F1000-0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000-0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000-0x101F3FFF	4KB

http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf

ELF Sections Example

```
$ arm-none-eabi-objdump -h example.elf
example.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000068  00010000  00010000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .ARM.attributes 00000016  00000000  00000000  00008068  2**0
                  CONTENTS, READONLY
```

ELF Header Example

```
$ arm-none-eabi-objdump -f example.elf

example.elf:     file format elf32-littlearm
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00010000
```

ELF Symbol Table Example

```
$ arm-none-eabi-objdump -t example.elf
example.elf:     file format elf32-
littlearm
SYMBOL TABLE:
00010000 l    d  .text  00000000 .text
00010028 l       .text  00000000 rfib
00010024 l       .text  00000000 iloop
0001004c l       .text  00000000 rfib_exit
0001005c g       .text  00000000 _tests
00010000 g       .text  00000000 _start
```

local

global

Program starts at this address

Loading

- Get the binary loaded into memory and running
- More an operating systems concept
  - E.g., load an executable into memory and start it
  - Handled by QEMU for our purposes
    - Loads our binary starting at a particular memory address (`0x10000`)
    - Code at low, initial address (~`0x00000`) branches to that address

```
0x00000000:  e3a00000     mov  r0, #0       ; 0x0
0x00000004:  e59f1004     ldr  r1, [pc, #4]      ; 0x10
0x00000008:  e59f2004     ldr  r2, [pc, #4]      ; 0x14
0x0000000c:  e59ff004     ldr  pc, [pc, #4]      ; 0x18
0x00000010:  00000183
0x00000014:  0x000100
0x00000018:  0x010000    ; offset!
```

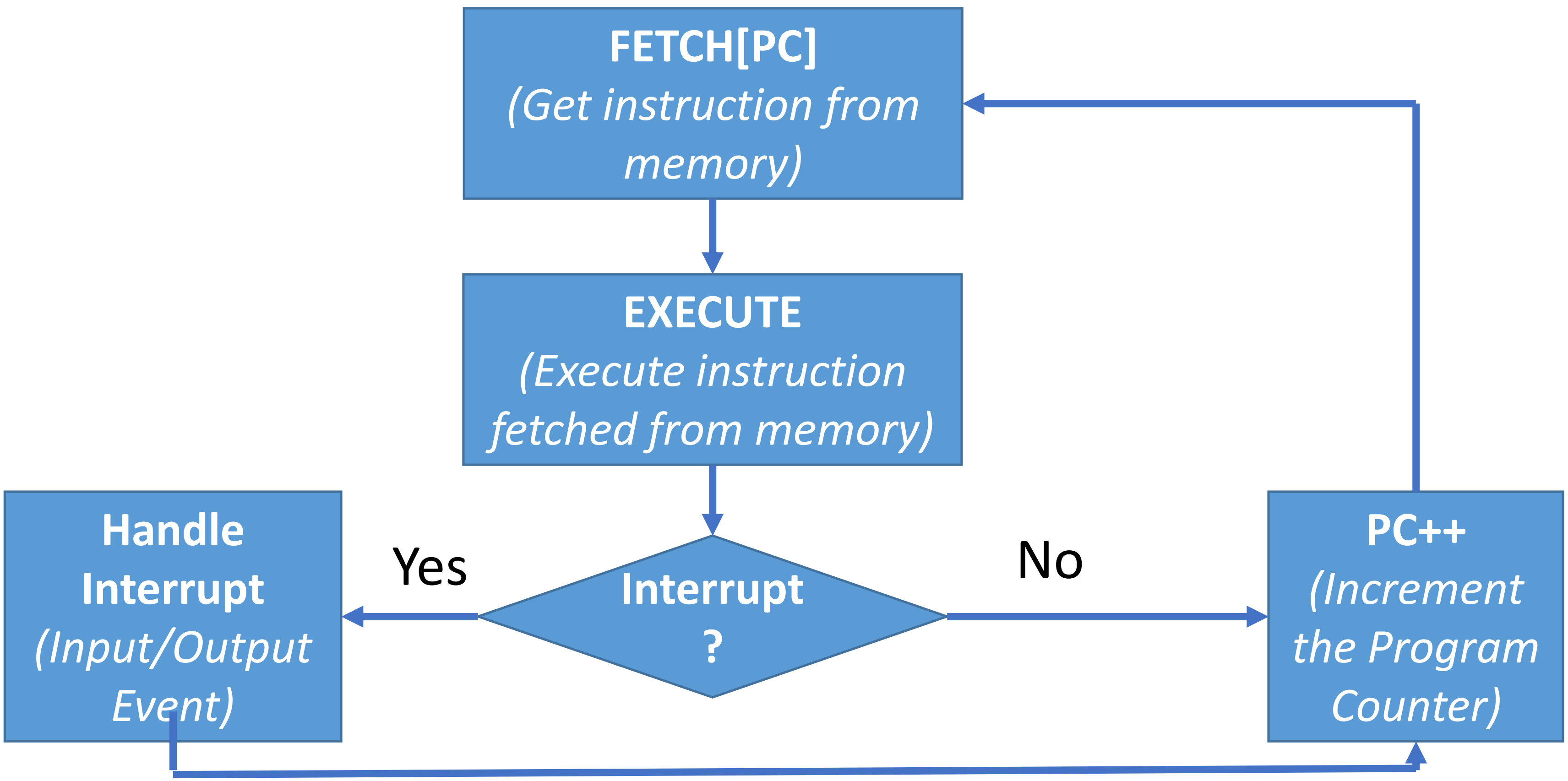
ARM 3 Stage Pipeline

- Stages: fetch, decode, execute
- PC value = instruction being fetched
- PC – 4: instruction being decoded
- PC – 8: instruction being executed

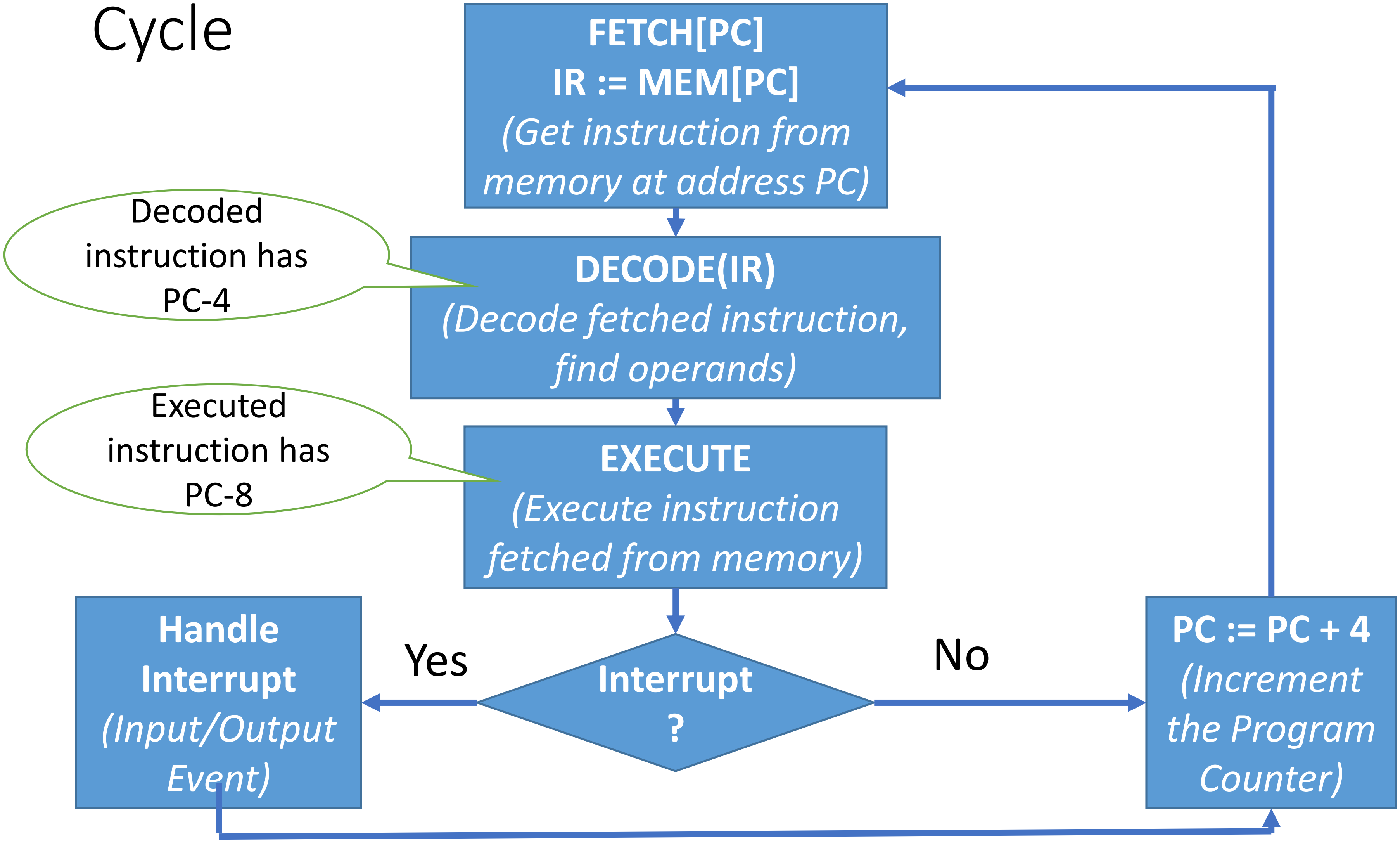
- Beefier ARM variants use deeper pipelines (5 stages, 13 stages)

UNIVERSITY OF TEXAS ARLINGTON

Recall: Abstract Processor Execution Cycle (*Simplified*)



ARM 3-Stage Pipeline Processor Execution Cycle



Summary

- Macros
- Assembly Process
- Memory Maps
  - Stack (data) location
  - Program location
  - Preview of memory-mapped I/O (device register location)
- Linking/Loading

UNIVERSITY OF TEXAS ARLINGTON

Questions?

?

String Output

- So far we have seen character input/output
- That is, one char at a time
- What about strings (character arrays, i.e., multiple characters)?
- Recall that strings are stored in memory at consecutive addresses

```
string_abc:
.asciz "abcdefghijklmnopqrstuvwxyz\n\r"
.word 0x00
```

ADDR	Byte 3	Byte 2	Byte 1	Byte 0
0x1000	'd'	'c'	'b'	'a'
0x1004	'h'	'g'	'f'	'e'
0x1008	'l'	'k'	'j'	'i'
0x100c	'p'	'o'	'n'	'm'
0x1010	't'	's'	'r'	'q'
0x1014	'x'	'w'	'v'	'u'
0x1018	'\0'	'\0'	'z'	'y'

Assembler Output

```
0001012e <string_abc>:
   1012e: 64636261   strbtvs    r6, [r3], #-609; 0x261
   10132: 68676665   stmdavs    r7!, {r0, r2, r5, r6, r9, sl, sp, 
lr}^
   10136: 6c6b6a69   stclvs     10, cr6, [fp], #-420; 0xfffffe5c
   1013a: 706f6e6d   rsbvc      r6, pc, sp, ror #28
   1013e: 74737271   ldrbtvc    r7, [r3], #-625; 0x271
   10142: 78777675   ldmdavc    r7!, {r0, r2, r4, r5, r6, r9, sl, 
ip, sp, lr}^
   10146: 0d0a7a79   vstreq     s14, [sl, #-484]     ; 0xfffffe1c
   1014a: 00000000   andeq      r0, r0, r0
```

ASCII

Binary	Octal	Decimal	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
…				…
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain first character of string to display
print_string: push  {r1,r2,lr}
str_out: ldrb  r2,[r1]
     cmp  r2,#0x00  @ '\0' = 0x00: null character?
     beq  str_done  @ if yes, quit
     str  r2,[r0]   @ otherwise, write char of string
     add  r1,r1,#1  @ go to next character
     b    str_out   @ repeat
str_done: pop  {r1,r2,lr}
     bx   lr
```