UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 15: Running ARM Programs in QEMU and Debugging with gdb

Taylor Johnson

Announcements and Outline

- Homework 5 due Thursday
- Midterm results

- Running ARM assembly programs with QEMU and debugging with gdb

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

Assembling ARM Programs

- How is this done?
  - 2-pass assembler process described before
- How is this done in practice?
  - Use an assembler like gcc's as
- Like with C programs, call 'make'
- What does this do?
  - Calls a command script specified in the file 'Makefile'

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
```

Linking and Loading

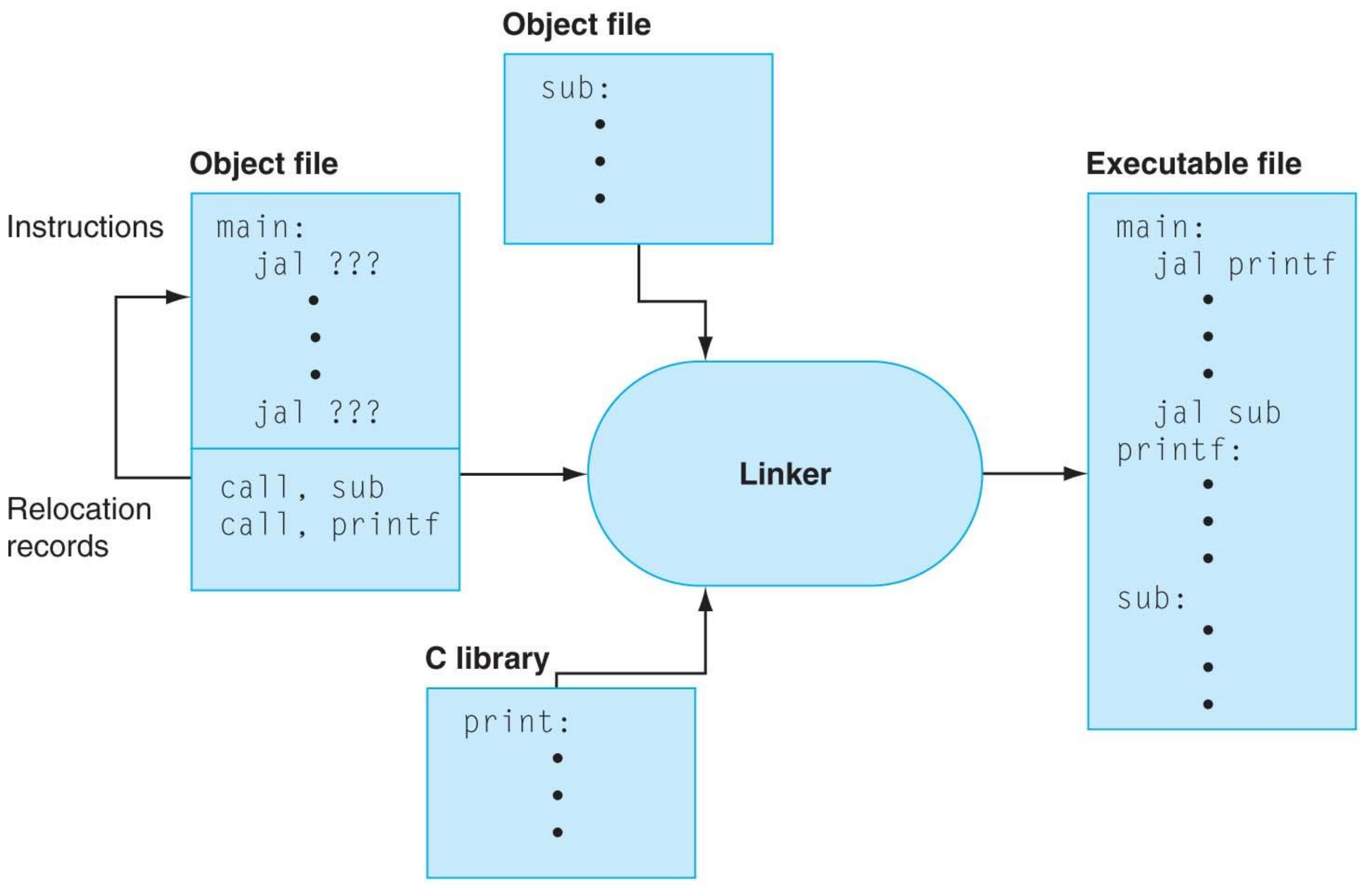
- Linking: combining multiple program modules (pieces of code) into executable program
  - Examples: using our _tests files to load inputs to your programs, calling library functions like printf, etc.
- Loading: getting executable running on machine
  - Examples: calling QEMU with our binary
- Static 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

Linker

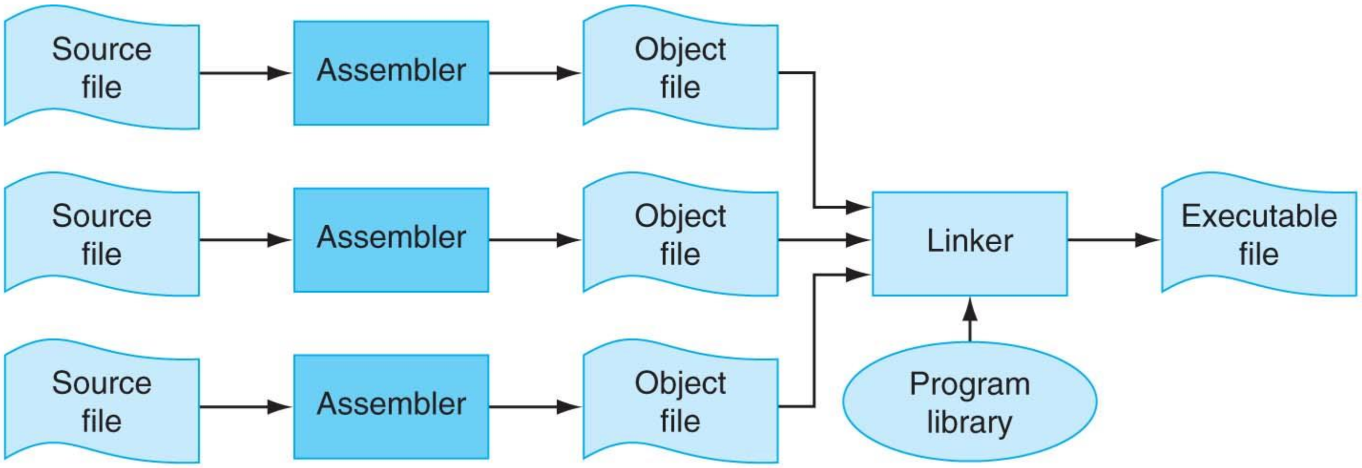
- `ld`
  - For us: `arm-none-eabi-ld`
  - The GNU ARM linker
- Operations
  - Copy code from each input file into resulting binary
  - Resolve references between files
  - Relocate symbols to use absolute memory addresses instead of relatives
  - Binary format

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

Linker Process



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.

Review: 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
```

Review: 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

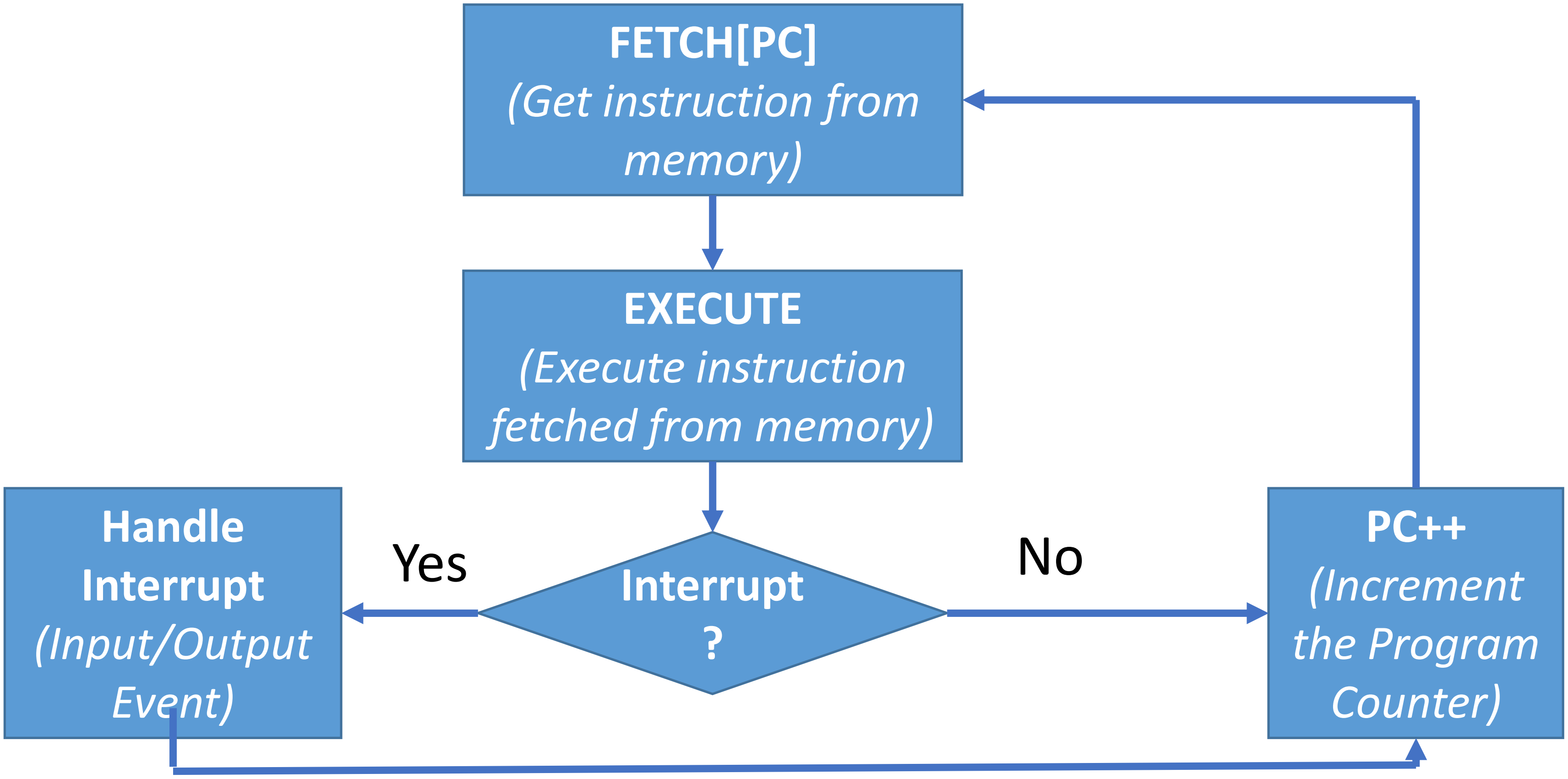
Program starts at this address

Loading

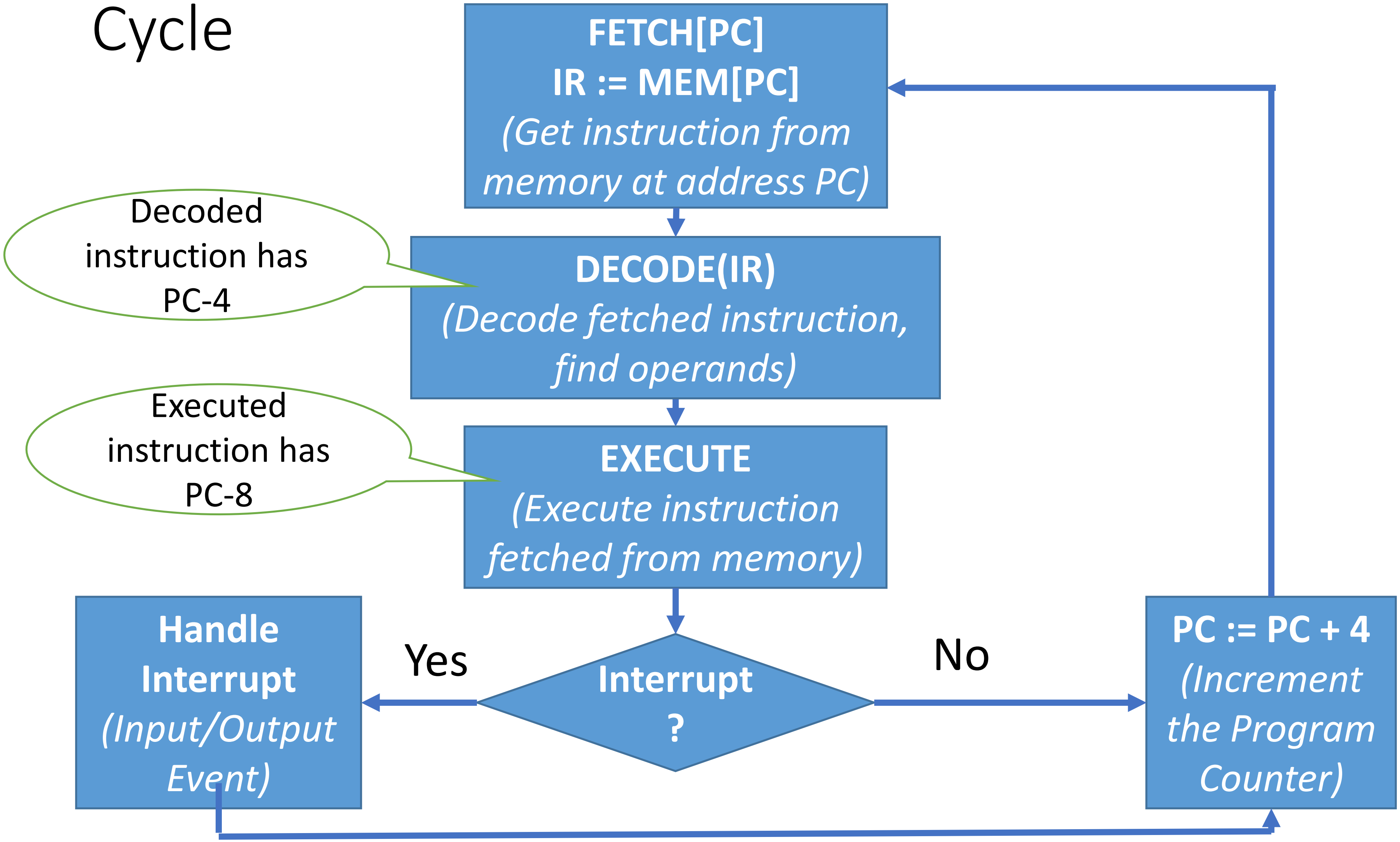
- Get the binary loaded into memory and running
- More an operating systems concept
  - E.g., load an executable into memory and start it
  - Handled by QEMU for our purposes
    - Loads our binary starting at a particular memory address (`0x10000`)
    - Code at low, initial address (~`0x00000`) branches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

Review: Abstract Processor Execution Cycle



ARM 3-Stage Pipeline Processor Execution Cycle



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)

QEMU

- Virtual machine: Quick-Emulator: http://www.qemu.org
- "QEMU is a generic and open source machine emulator and virtualizer."
- "When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance."
- "When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests."
- ***<u>QEMU runs like any other Linux process/program</u>***

Starting and Finishing QEMU

- **Start command (1 line):**

```
qemu-system-arm -S -s -M versatilepb -daemonize -m 128M -d
in_asm,cpu,exec -kernel example.bin
```

- **Make sure QEMU actually quits when done (it's a daemonized process!):**

```
ps aux | grep qemu
```

```
> tjohnson 14437  3.0  0.1 401572 10240 ?        Sl   11:27   0:09 qemu-system-arm -S -s -M
versatilepb -daemonize -m 128M -d in_asm,cpu,exec -kernel example.bin
```

```
> kill -9 14437
```

- Process id is 14437
- **Lots of options for qemu, do:**

```
qemu-system-arm --help
```

Starting GDB

- GDB is another process
- Interacts with QEMU process
- Start via:

```
gdb-multiarch --init-command=.gdbinit
```

- The --init-command=.gdbinit may be optional (based on system configuration)
- Saves you time, executes commands in .gdbinit before starting gdb
- Lots of other options, do:

```
gdb-multiarch --help
```

Example .gdbinit

```
set architecture arm
target remote :1234
symbol-file example.elf
b _start
Pause
```

- Sets architecture to arm (default is x86)
- Connects to QEMU process via port 1234
- Loads symbols (labels, etc.) from the ELF file called example.elf
- Puts breakpoint at label _start
- Pauses execution (to start up stopped)

GDB Commands

- `b label`
  Sets a breakpoint at a specific label in your source code file. In practice, for some weird reason, the code actually breaks not at the label that you specify, but after executing the next line.
- `b line_number`
  Sets a breakpoint at a specific line in your source code file. In practice, for some weird reason, the code actually breaks not at the line that you specify, but at the line right after that.
- `c`
  Continues program execution until it hits the next breakpoint.
- `i r`
  Shows the contents of all registers, in both hexadecimal and decimal representations; short for `info registers`
- `list`
  Shows a list of instructions around the line of code that is being executed.
- `quit`
  This command quits the debugger, and exits GDB.
- `stepi`
  This command executes the next instruction.
- `set $register=val`
  `set $pc=0`
  This command updates a register to be equal to val, for example, to restart your program, set the PC to 0

Basic Function Call Example

```
int ex(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

```
r0 = g, r1 = h, r2 = I, r3 = j, r4 = f
```

Basic Function Call Example Assembly

```
ex:                  ; label for function name
SUB sp, sp, #12   ; adjust stack to make room for 3 items
STR r6, [sp,#8]   ; save register r6 for use afterwards
STR r5, [sp,#4]   ; save register r5 for use afterwards
STR r4, [sp,#0]   ; save register r4 for use afterwards

ADD r5,r0,r1     ; register r5 contains g + h
ADD r6,r2,r3     ; register r6 contains i + j
SUB r4,r5,r6     ; f gets r5 - r6, ie: (g + h) - (i + j)
MOV r0,r4        ; returns f (r0 = r4)

LDR r4, [sp,#0] ; restore register r4 for caller
LDR r5, [sp,#4] ; restore register r5 for caller
LDR r6, [sp,#8] ; restore register r6 for caller
ADD sp,sp,#12    ; adjust stack to delete 3 items
MOV pc, lr       ; jump back to calling routine
```

Basic Function Call Example Call

```
Breakpoint  3, ex () at
example.s:17
17        ADD r5,r0,r1
(gdb) i r
r0              0x5    5
r1              0x4    4
r2              0x6    6
r3              0x7    7
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              0xfff0
    0xfff0
lr              0x1001c
    65564
pc              0x10024
    0x10024 <ex+4>
cpsr
0x400001d3
    1073742291
```

Basic Function Output

```
r0      0xfffffffc      -4
r1      0x4     4
r2      0x6     6
r3      0x7     7
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      0x1001c 65564
pc      0x1001c 0x1001c <iloop>
cpsr    0x400001d3      1073742291
```

```
@ (g + h) - (i + j)
@ r0 = g
@ r1 = h
@ r2 = i
@ r3 = j
@ r4 = f
        mov r0,#5
        mov r1,#4
        mov r2,#6
        mov r3,#7
        mov r4,#0
```

Basic Function Call Example Stack

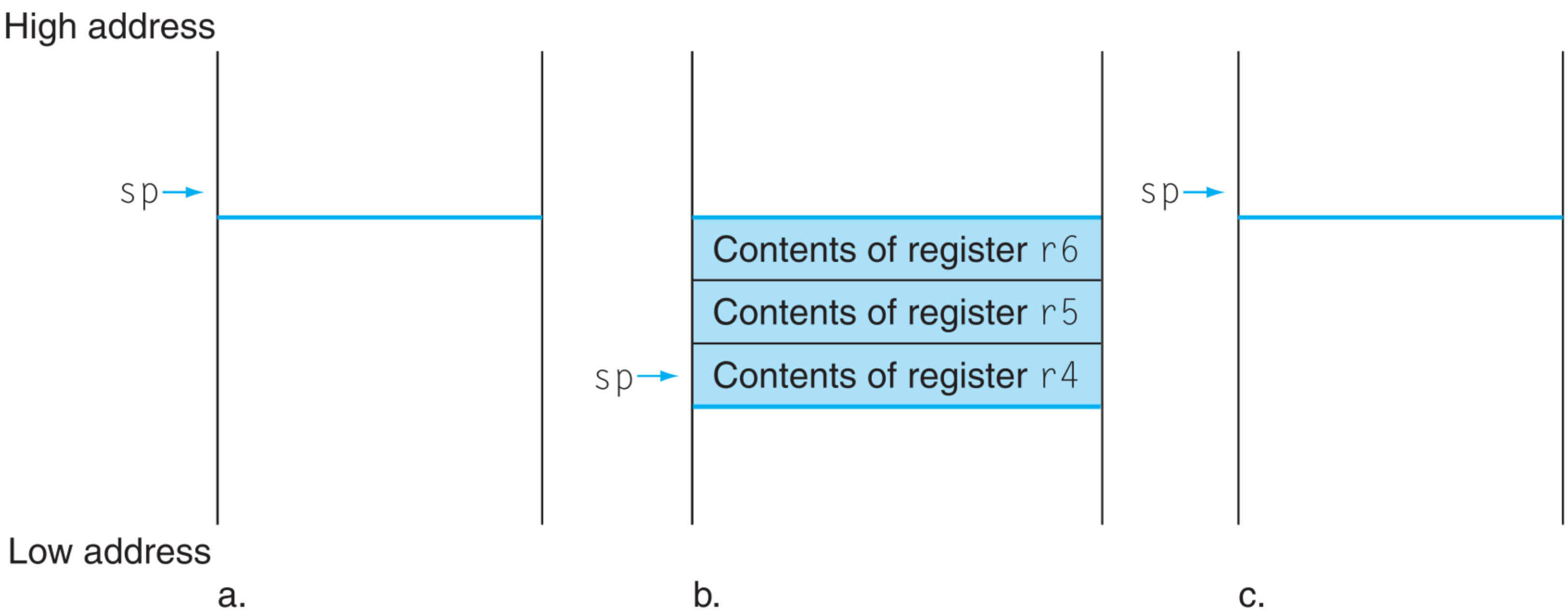


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Basic Function Call Example Assembly (Push/Pop)

```
ex:                 ; label for function name
PUSH {r4,r5,r6}  ; save r4, r5, r6, decrement sp by 12

ADD r5,r0,r1    ; register r5 contains g + h
ADD r6,r2,r3    ; register r6 contains i + j
SUB r4,r5,r6    ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4       ; returns f (r0 = r4)

POP {r4,r5,r6} ; restore r4, r5, r6, increment sp by 12
MOV pc, lr      ; jump back to calling routine
```

Recursive Function Example: Factorial

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

```
int factorial(int N)
{
  if (N== 0) return 1;
  return N* factorial(N -1);
}
```

- How do we write function factorial in assembly?

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

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

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

Summary

- Know what make does
- Know how to start QEMU
- Know how to start GDB
- Start learning how to interact and debug with GDB

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)?

- Strings are stored in memory at consecutive addresses
  - Like arrays that we saw last time

```
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	'\r'	'\n'	'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
```

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain address of first character of string
@ to display; stop if 0x00 ('\0') seen
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
```

Gdb: printing code to be executed

```
(gdb) x /16i $pc
=> 0x10008 <loop>:    add    r1, r1, #1
   0x1000c <loop+4>:  and    r1, r1, #7
   0x10010 <loop+8>:  add    r1, r1, #48   ; 0x30
   0x10014 <loop+12>:        str    r1, [r0]
   0x10018 <loop+16>:        mov    r2, #13
   0x1001c <loop+20>:        str    r2, [r0]
   0x10020 <loop+24>:        mov    r2, #10
   0x10024 <loop+28>:        str    r2, [r0]
   0x10028 <loop+32>:        b      0x10008 <loop>
   0x1002c <infloop>:        b      0x1002c <infloop>
   0x10030 <val>:     andeq  r0, r0, r1, lsl r0
   0x10034 <val+4>:   andeq  r0, r0, r2, lsr #32
   0x10038 <val+8>:   andeq  r0, r0, r3, lsr r0
   0x1003c <val+12>:  andeq  r0, r0, r4, asr #32
   0x10040 <val+16>:  andeq  r0, r0, r5, asr r0
   0x10044 <val+20>:  andeq  r0, r0, r6, rrx
(gdb)
```

Questions?

?