UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 16: Processor Pipeline Introduction and Debugging with GDB

Taylor Johnson

Announcements and Outline

- Homework 5 due today
  - Know how to assemble/link programs, start them in QEMU, and start debugging them with gdb
  - Start to learn how to use gdb

- Running ARM assembly programs with QEMU and debugging with gdb
  - Debugging a basic procedure and looking at the statck
  - Debugging a recursive procedure and looking at the stack

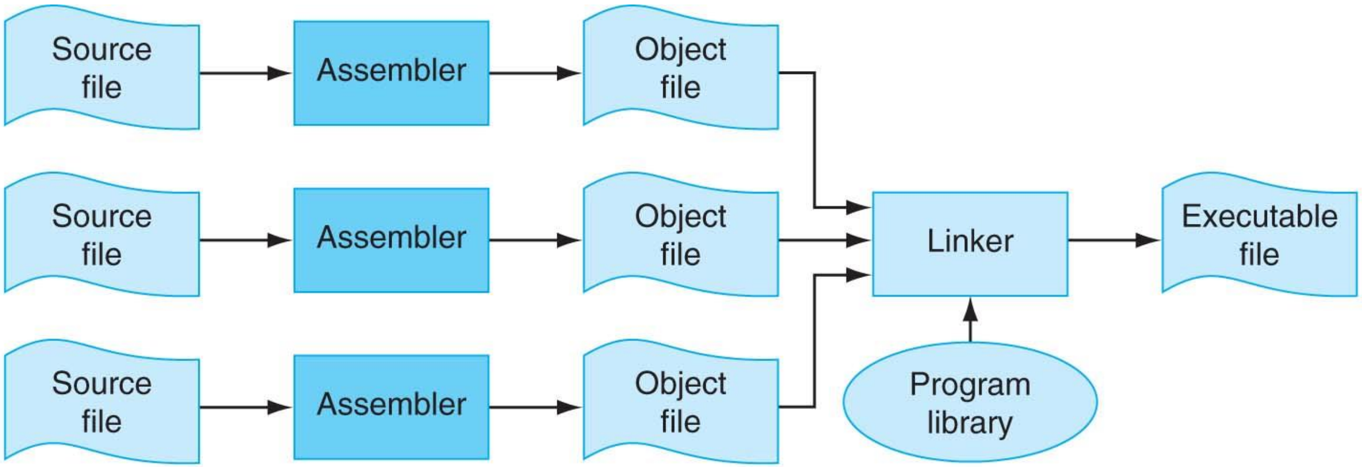
Review: Assembling ARM Programs

- 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’

Review: Makefile Example

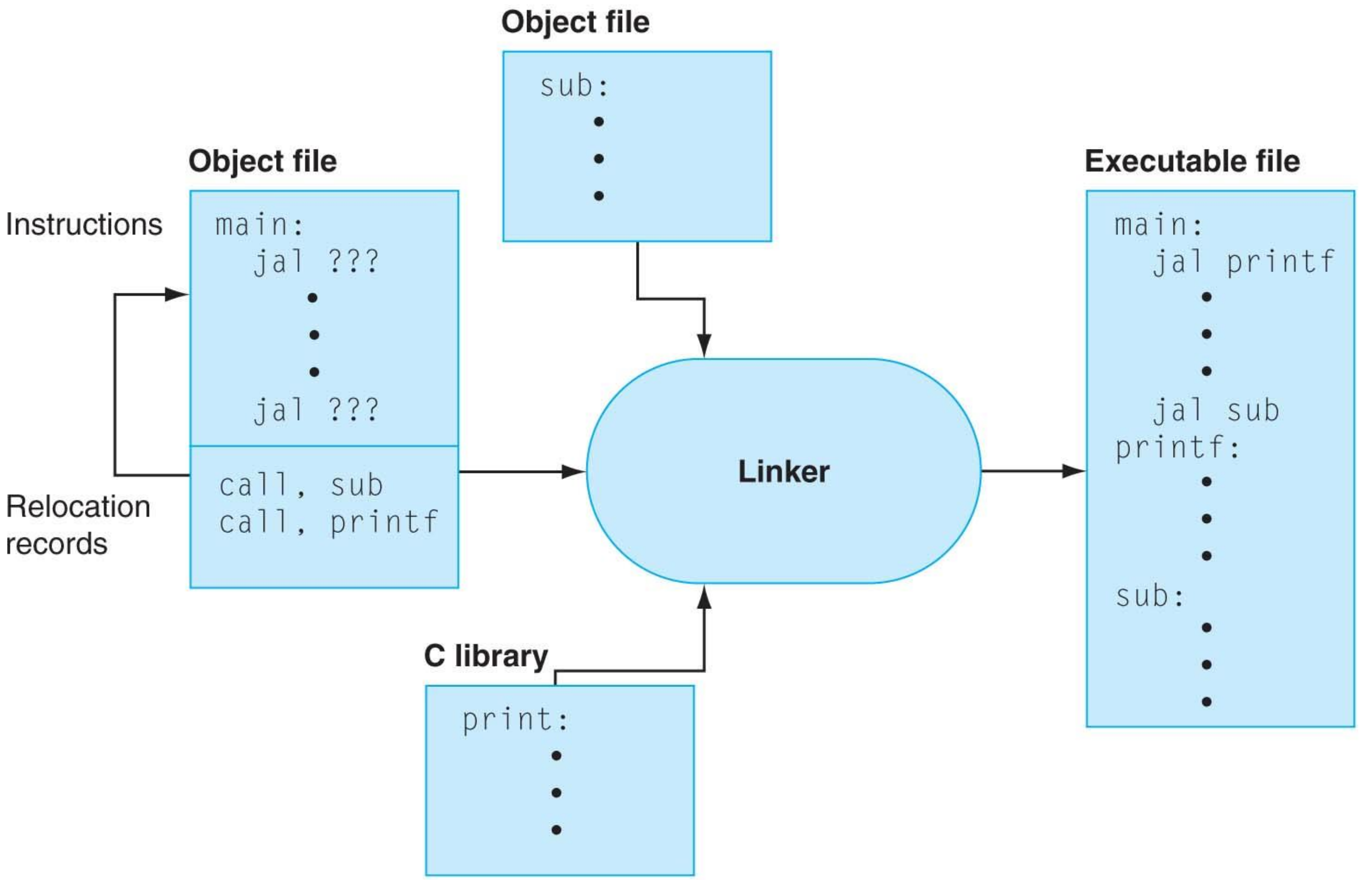
```
CROSS_COMPILE ?= arm-none-eabi
AOPS = --warn --fatal-warnings -g
example.bin : example.s example_tests.s example_memmap
     $(CROSS_COMPILE)-as $(AOPS) example.s -o example.o
     $(CROSS_COMPILE)-as $(AOPS) example_tests.s –o example_tests.o
     $(CROSS_COMPILE)-ld example.o example_tests.o -T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

Review: 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: Linker Process

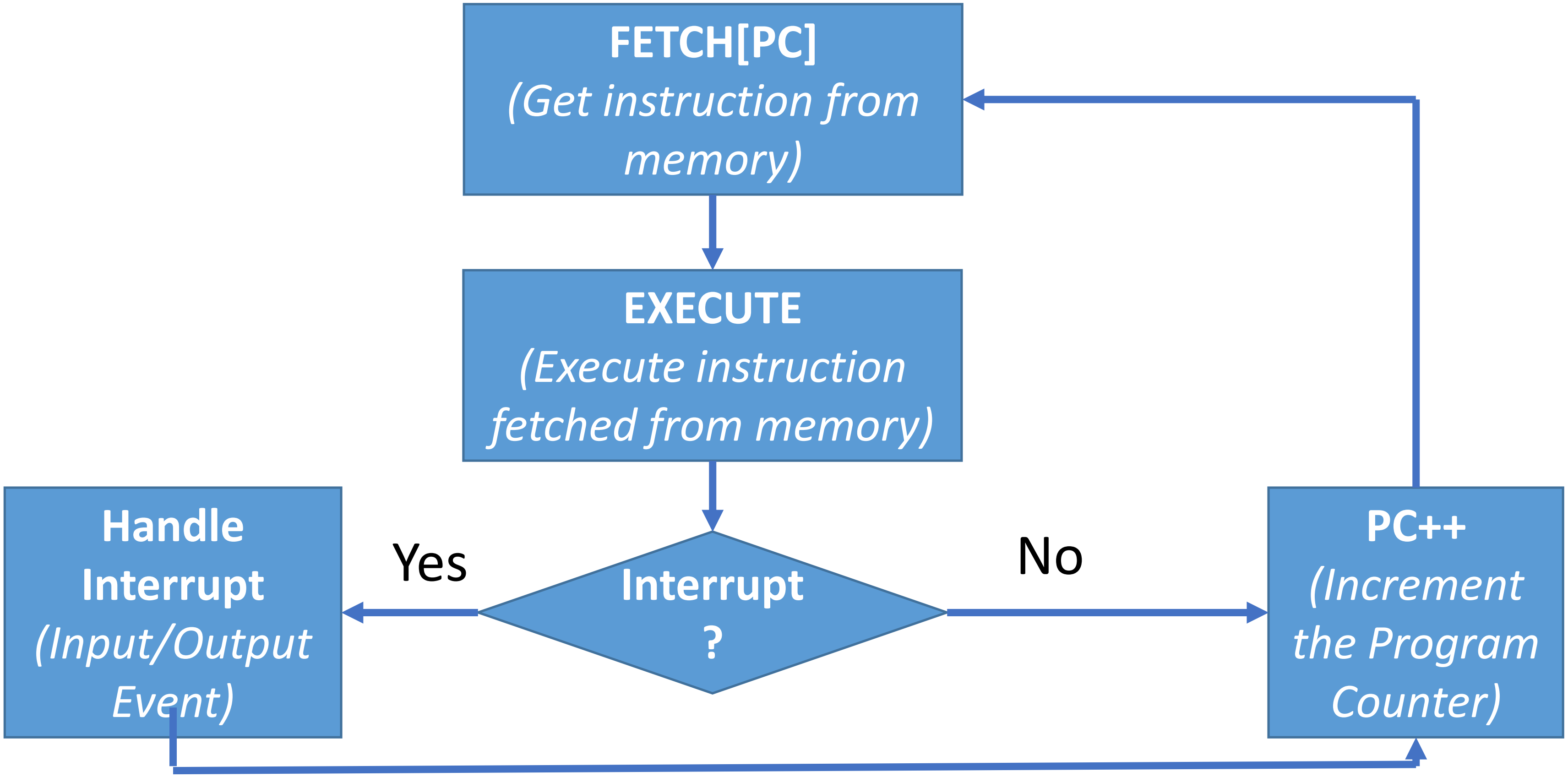


Review: Loading

- 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



FETCH[PC]
(Get instruction from memory)

EXECUTE
(Execute instruction fetched from memory)

Interrupt ?

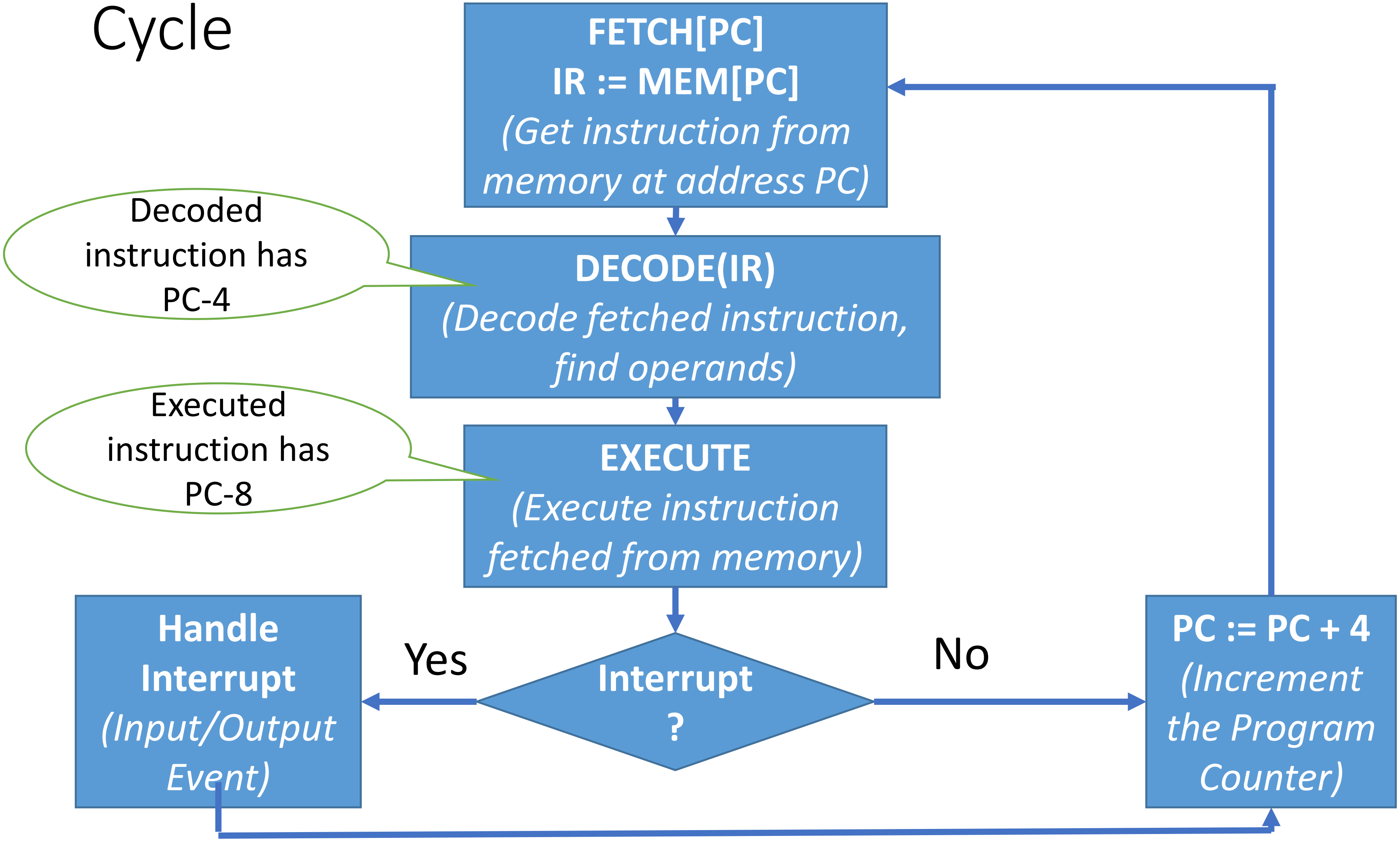
Yes

Handle Interrupt
(Input/Output Event)

No

PC++
(Increment the Program Counter)

ARM 3-Stage Pipeline Processor Execution Cycle



FETCH[PC]
IR := MEM[PC]
(Get instruction from memory at address PC)

DECODE(IR)
(Decode fetched instruction, find operands)

EXECUTE
(Execute instruction fetched from memory)

Interrupt ?

Yes

Handle Interrupt
(Input/Output Event)

No

PC := PC + 4
(Increment the Program Counter)

Decoded instruction has PC-4

Executed instruction has PC-8

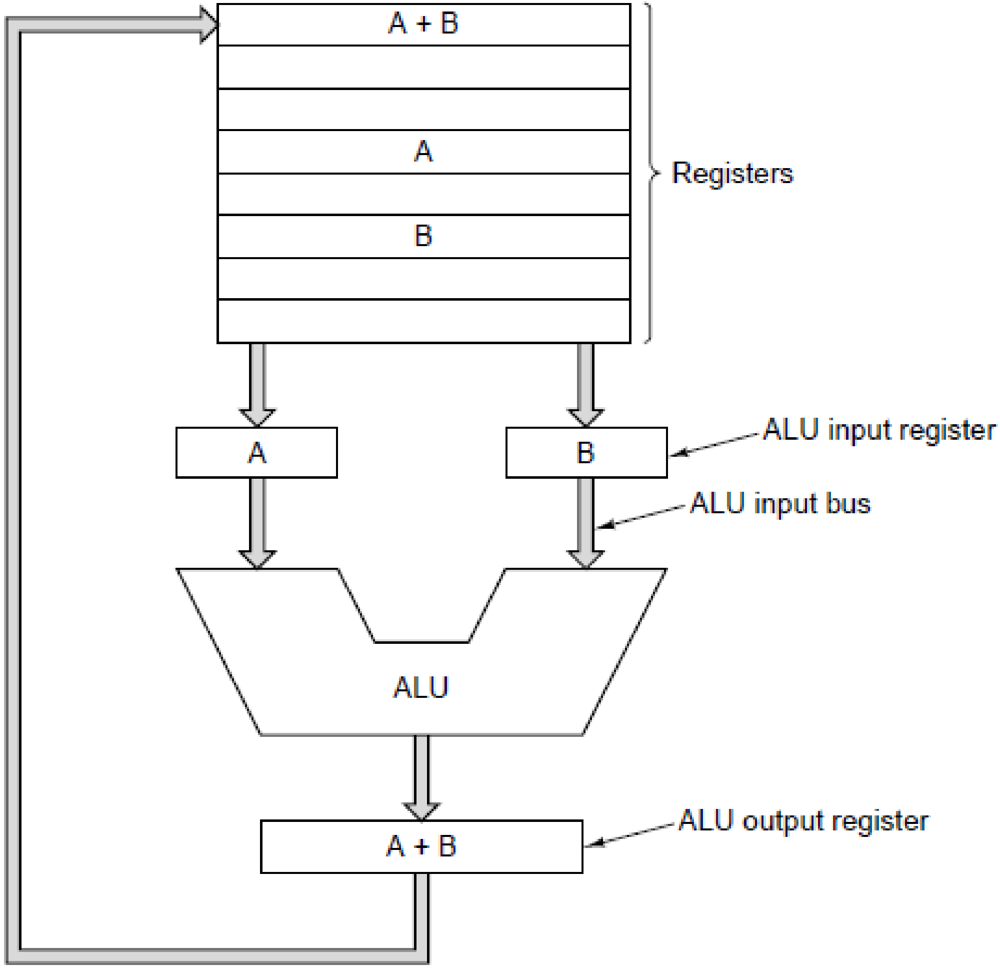
ARM 3 Stage Pipeline

- 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)

Data Path

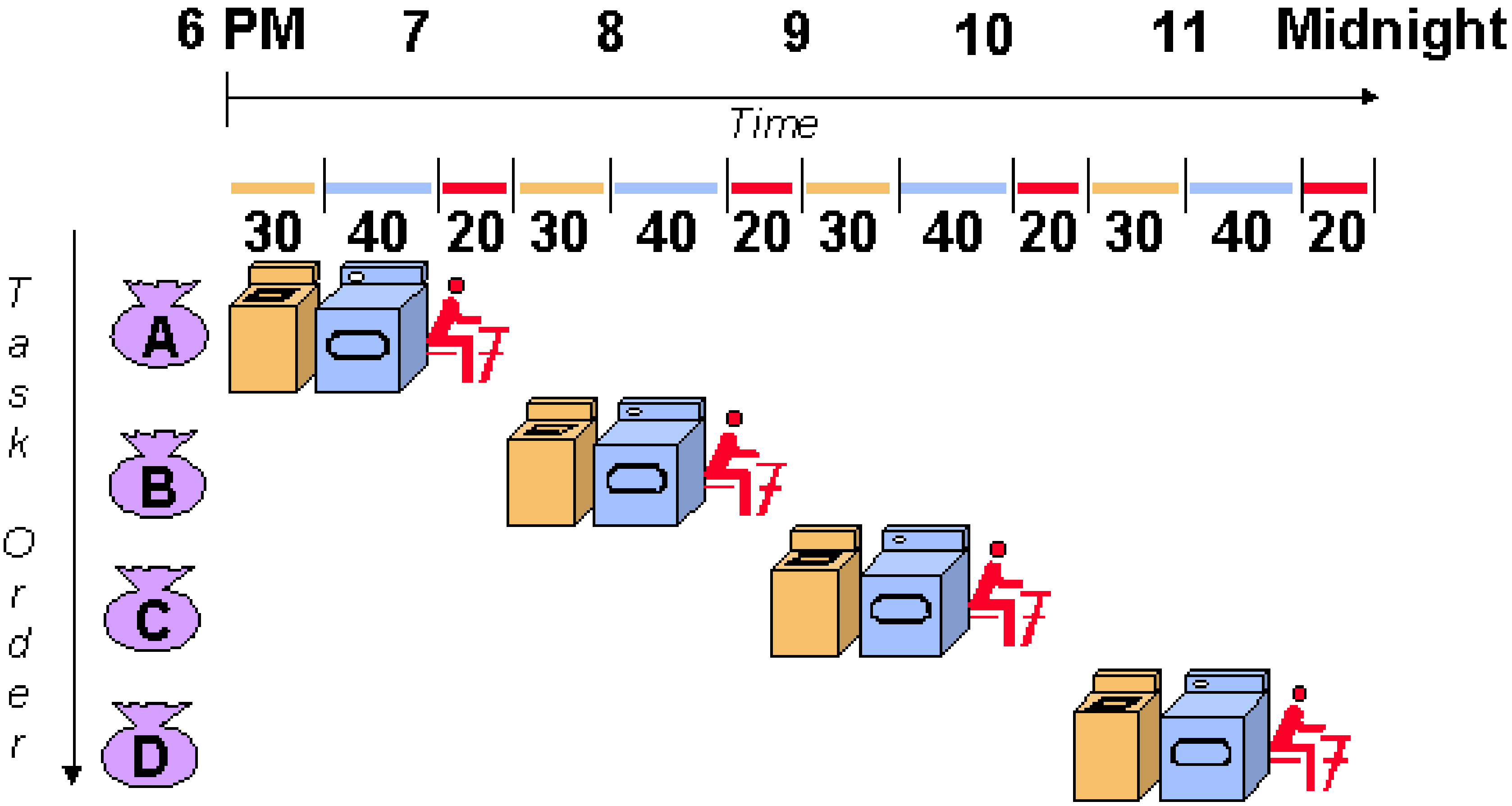
- Instructions
  - Register-Memory: memory words being fetched into registers
  - Register-Register
- Data Path Cycle
  - The process of running two operands through the ALU and storing results
  - Defines what the machine can do
  - The faster the data path cycles, the faster the computer



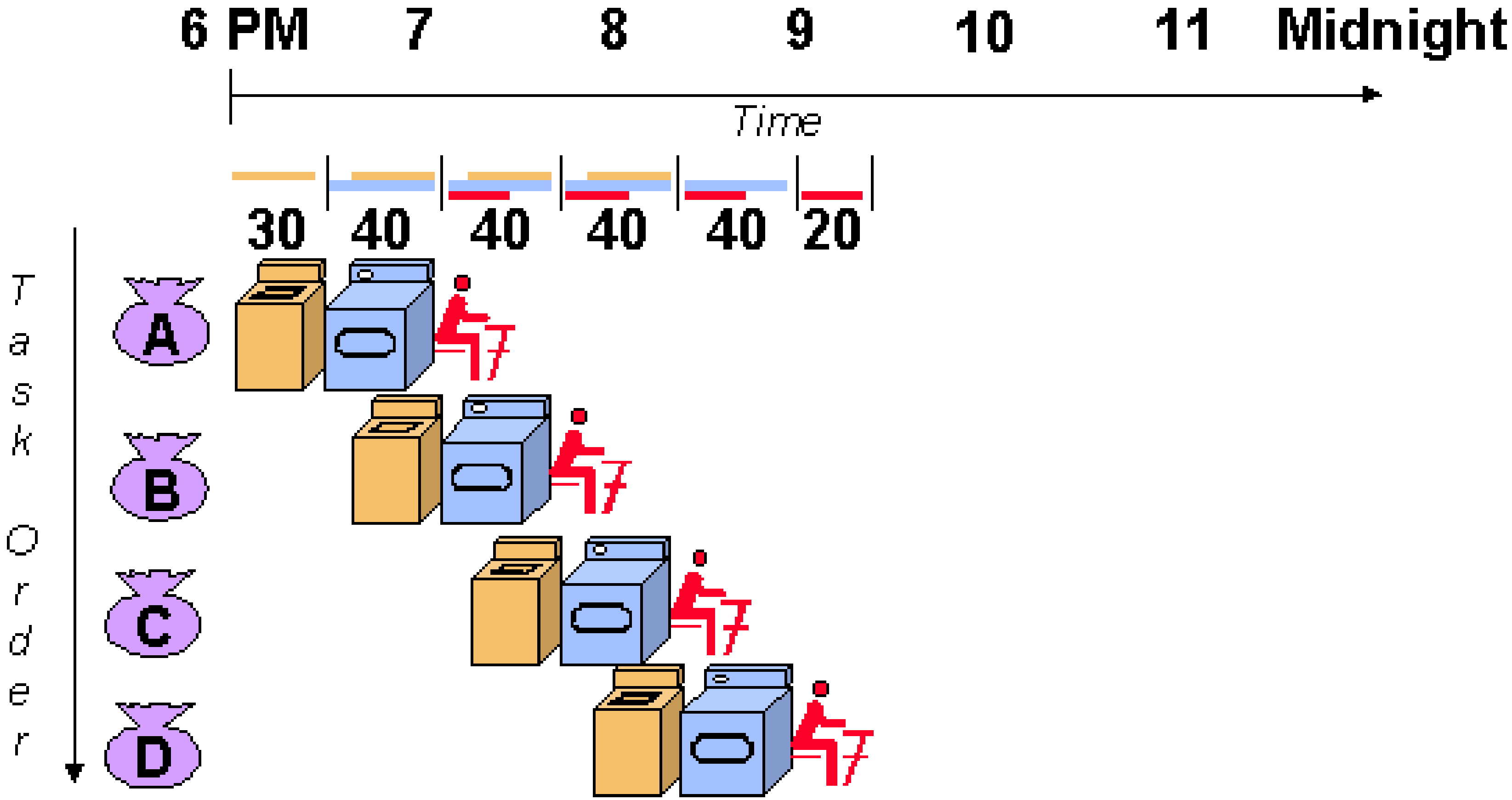
Instruction Execution

- Fetch next instruction from memory
- Change program counter to point to next instruction
- Determine type of instruction just fetched
- If instruction uses memory, locate it
- Fetch memory, if needed, into a CPU register
- Execute instruction
- Go to step 1 to begin executing following instruction

Un-Pipelined Laundry



Pipelined Laundry



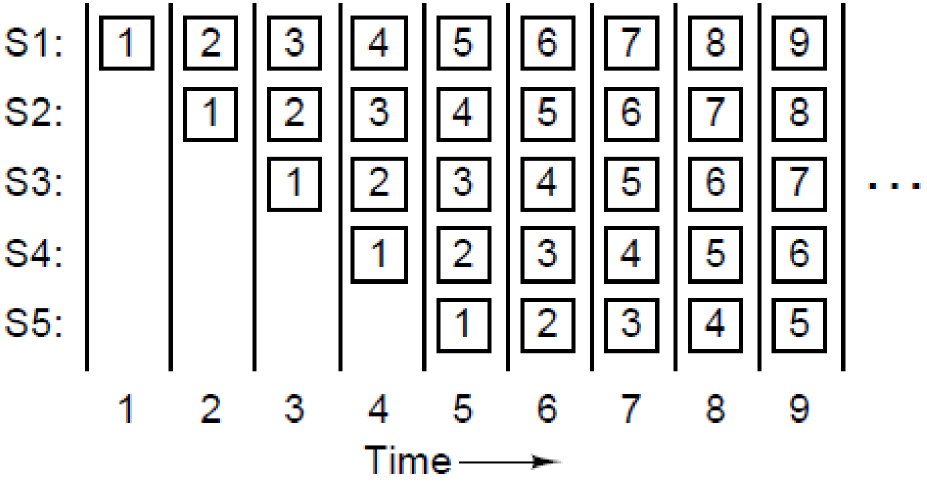
Why Pipelining?

- Consider a five-stage pipeline
  - Suppose 2ns for the cycle period
  - It takes 10ns for an instruction to progress all the way through pipeline
  - So, the machine runs at 100 MIPS?
  - Actual rate: 500 MIPS
- Pipelining
  - Tradeoff between latency and processor bandwidth
  - Latency: how long it takes to execute an instruction
  - Processor bandwidth: MIPS of the CPU
- Example
  - Suppose a complex instruction should take 10 ns, under perfect conditions, how many stage pipeline should we design to guarantee 500 MIPS?
  - Each pipeline stage should take: 1/500 MIPS = 2 ns
  - 10 ns/ 2ns =5 stages

Pipelining: Instruction-Level Parallelism

S1 Instruction fetch unit → S2 Instruction decode unit → S3 Operand fetch unit → S4 Instruction execution unit → S5 Write back unit

(a)

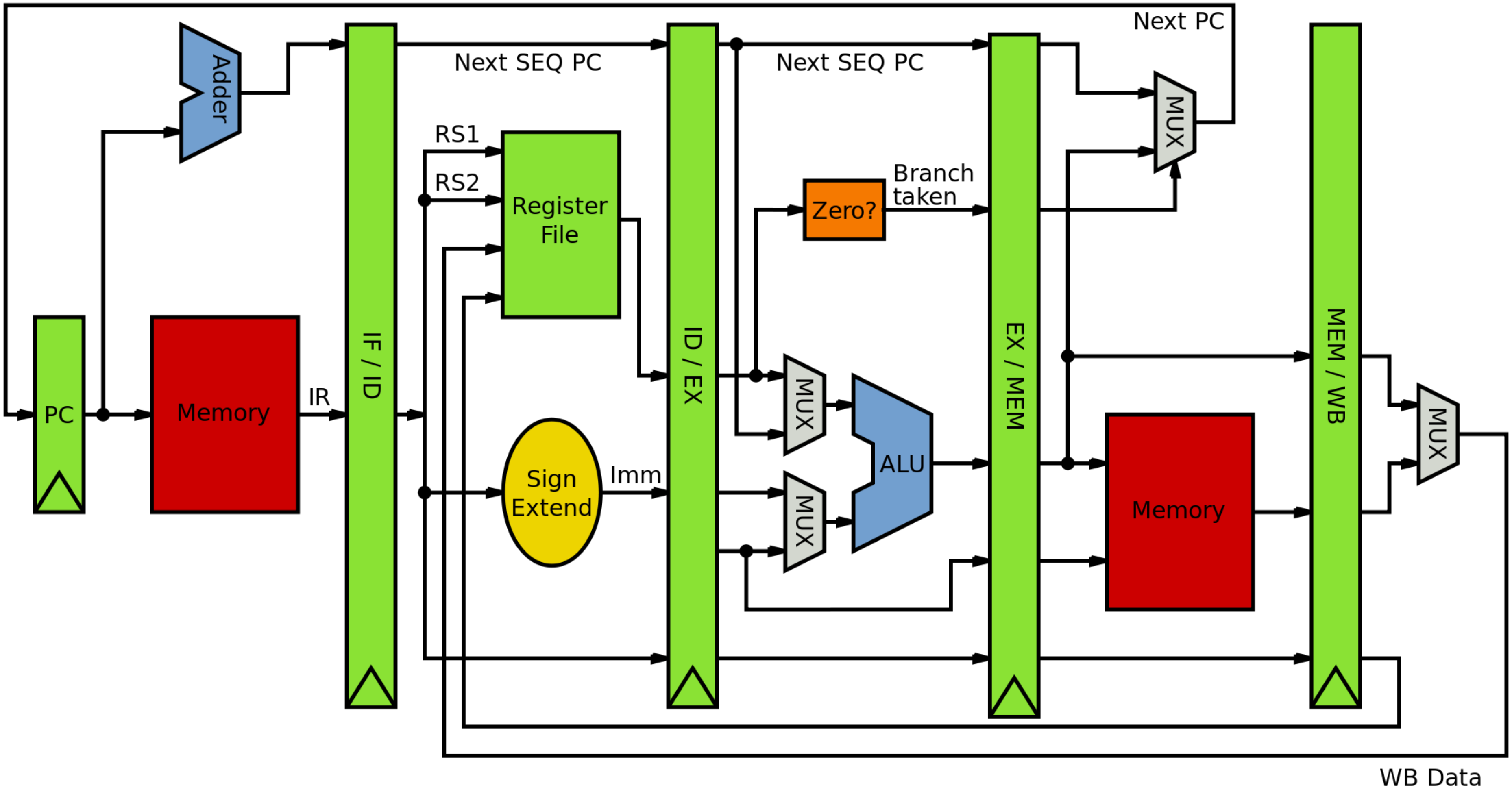


	1	2	3	4	5	6	7	8	9	
S1:	1	2	3	4	5	6	7	8	9	
S2:		1	2	3	4	5	6	7	8	
S3:			1	2	3	4	5	6	7	. . .
S4:				1	2	3	4	5	6	
S5:					1	2	3	4	5	

Time →

(b)

Instruction Fetch	Instruction Decode Register Fetch	Execute Address Calc.	Memory Access	Write Back
IF	ID	EX	MEM	WB

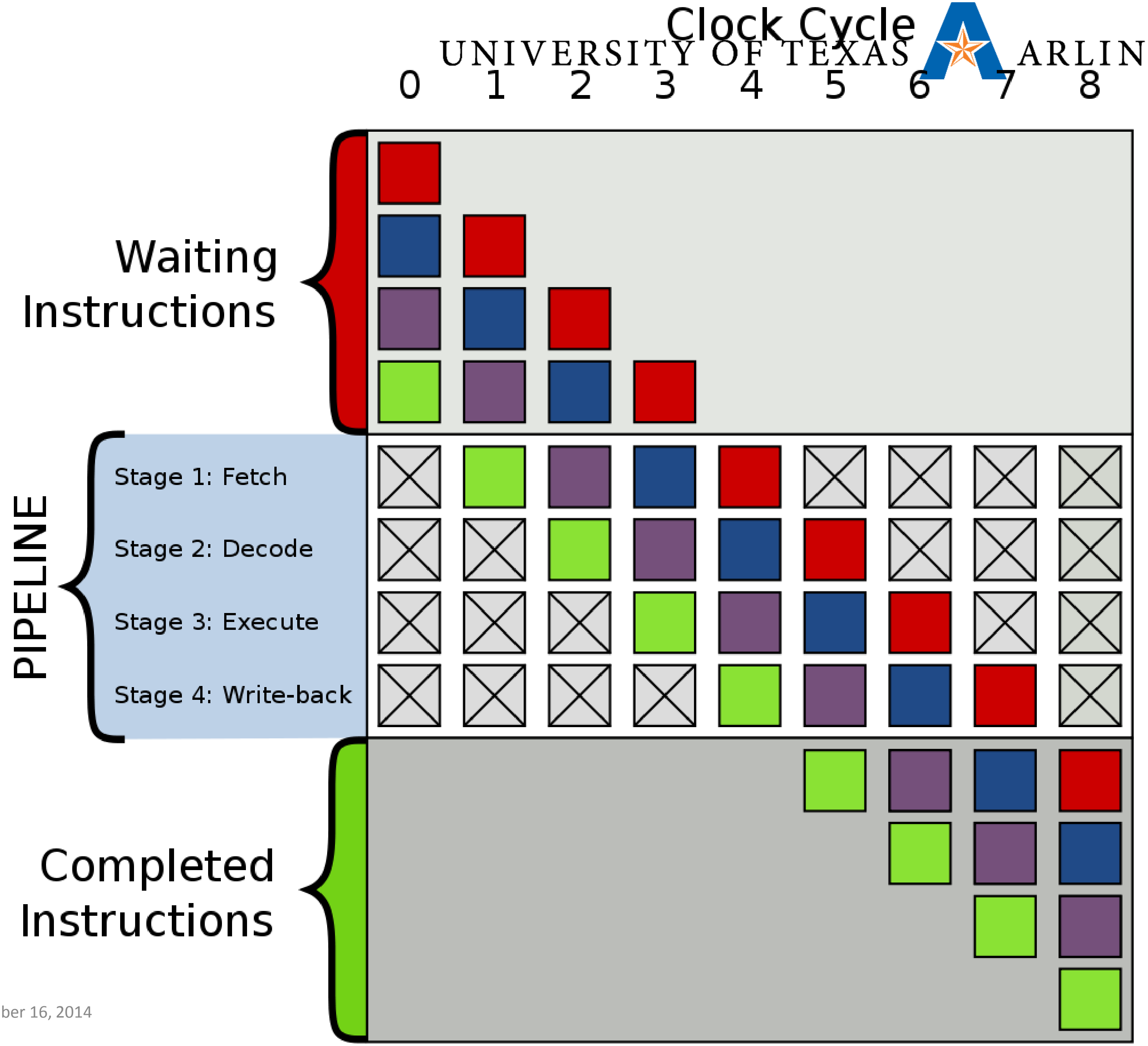


Hazards

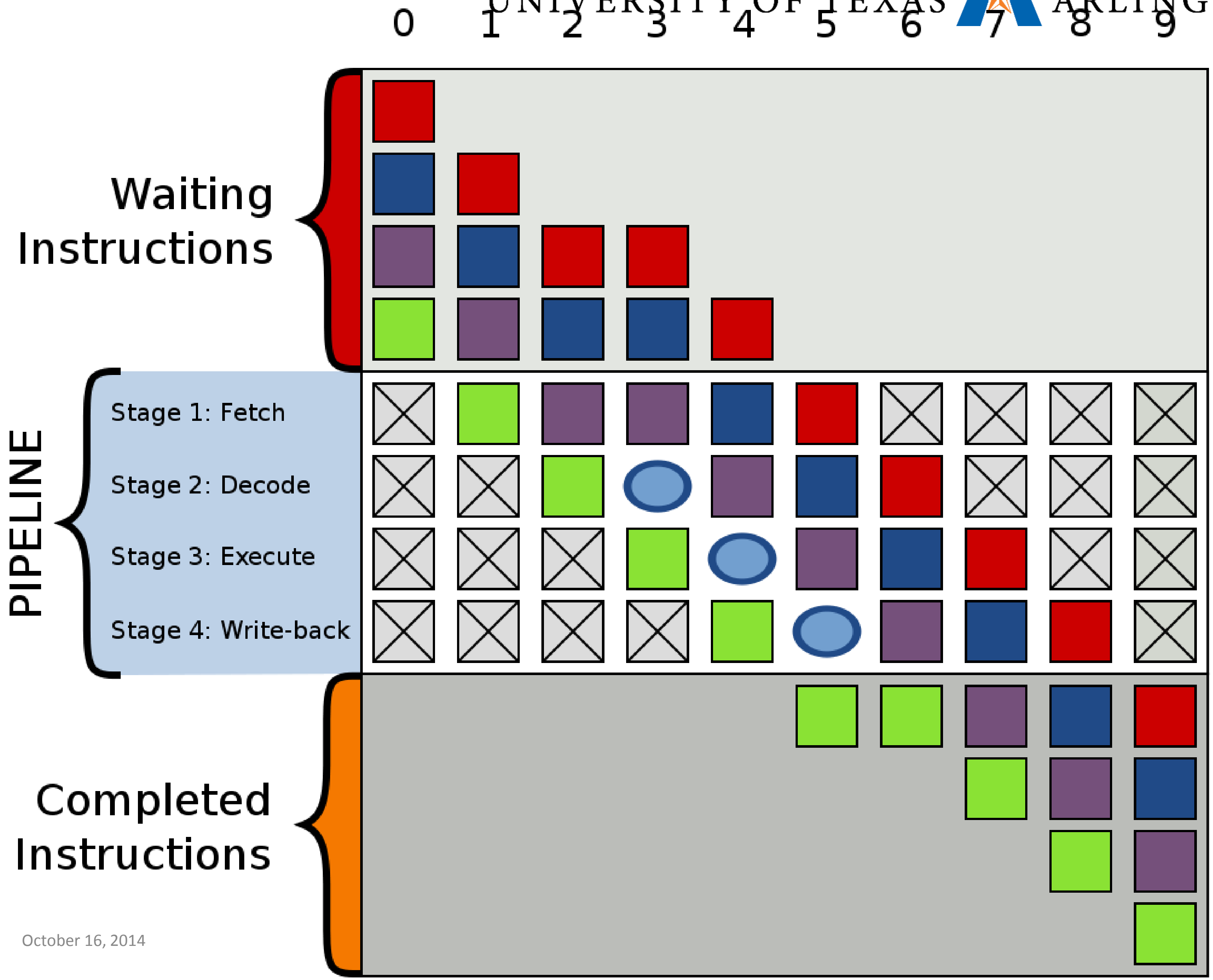
- Hazard: next instruction cannot execute in next cycle
- Data hazards: instruction depends on result of prior instruction
  - Example:

    ```
    store 0x1234 r0
    load r0 0x1234
    ```

    Problem: `load` cannot occur until `store` has completed
  - Solution: stall, out-of-order execution, register forwarding
- Structural Hazards:
  - Solution: stall
- Control Hazards: direction of control flow (e.g., branch) depends on prior instructions
  - Solution: stall, branch prediction


Clock Cycle
0
1
2
3
4
5
6
7
8
Waiting Instructions
PIPELINE
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed Instructions

Clock Cycle



Waiting Instructions

PIPELINE

Stage 1: Fetch

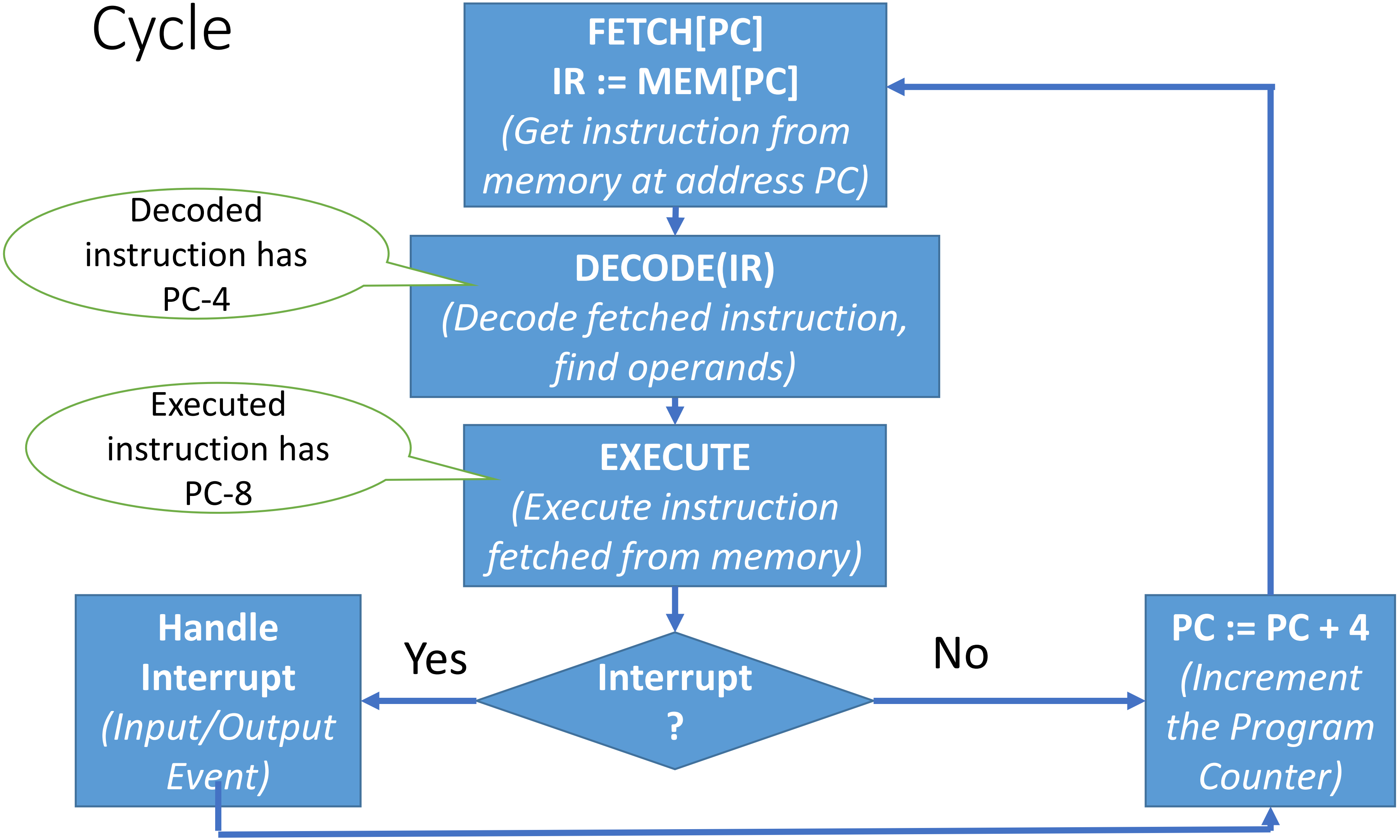
Stage 2: Decode

Stage 3: Execute

Stage 4: Write-back

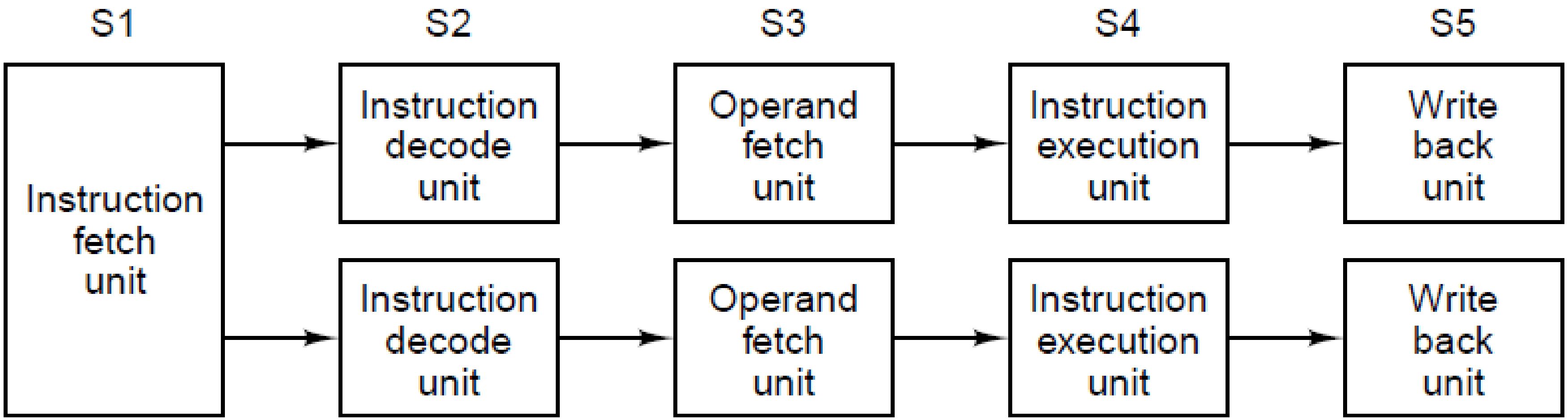
Completed Instructions

ARM 3-Stage Pipeline Processor Execution Cycle



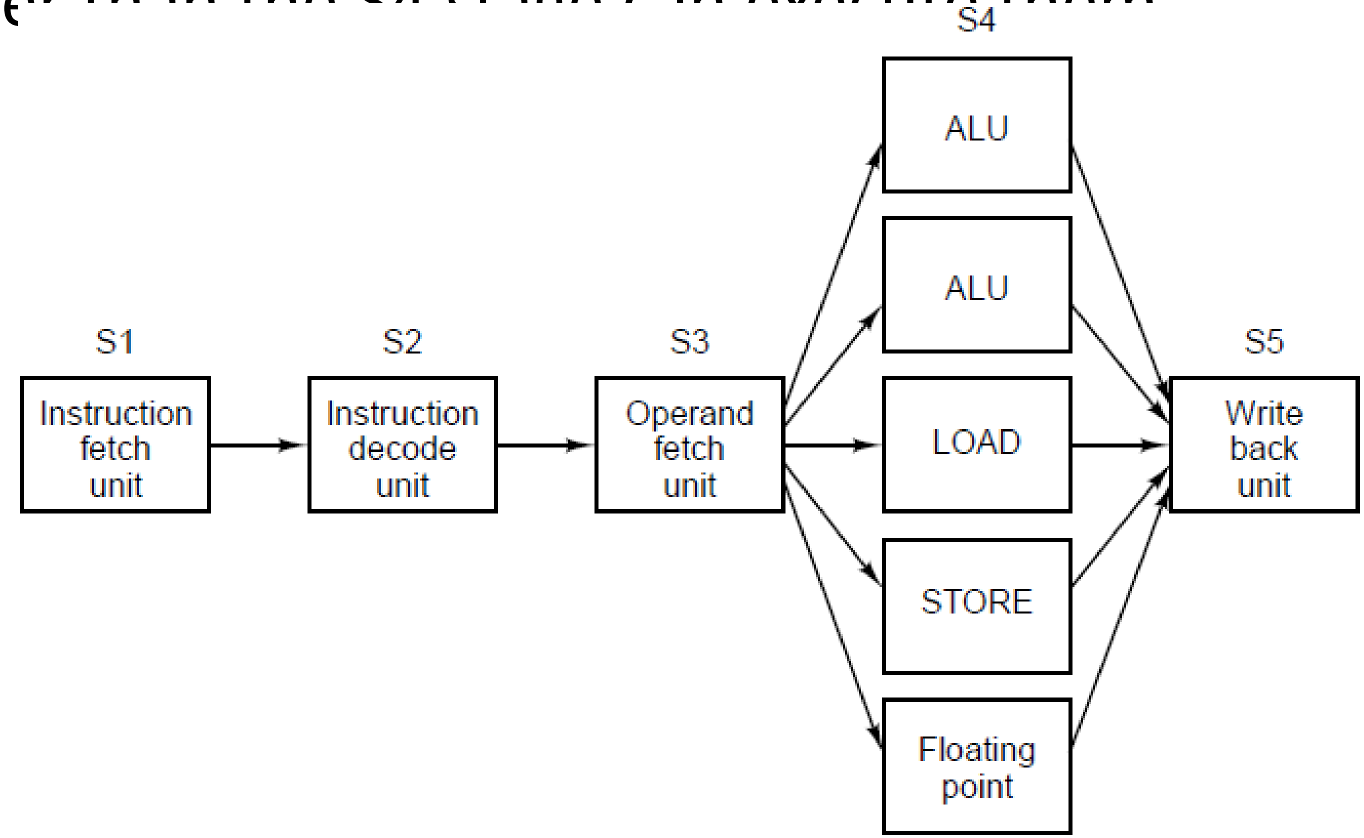
Superscalar Architectures

- Dual five-stage pipelines with common instruction fetch unit
  - Fetches pairs of instructions together and puts each into its own pipeline

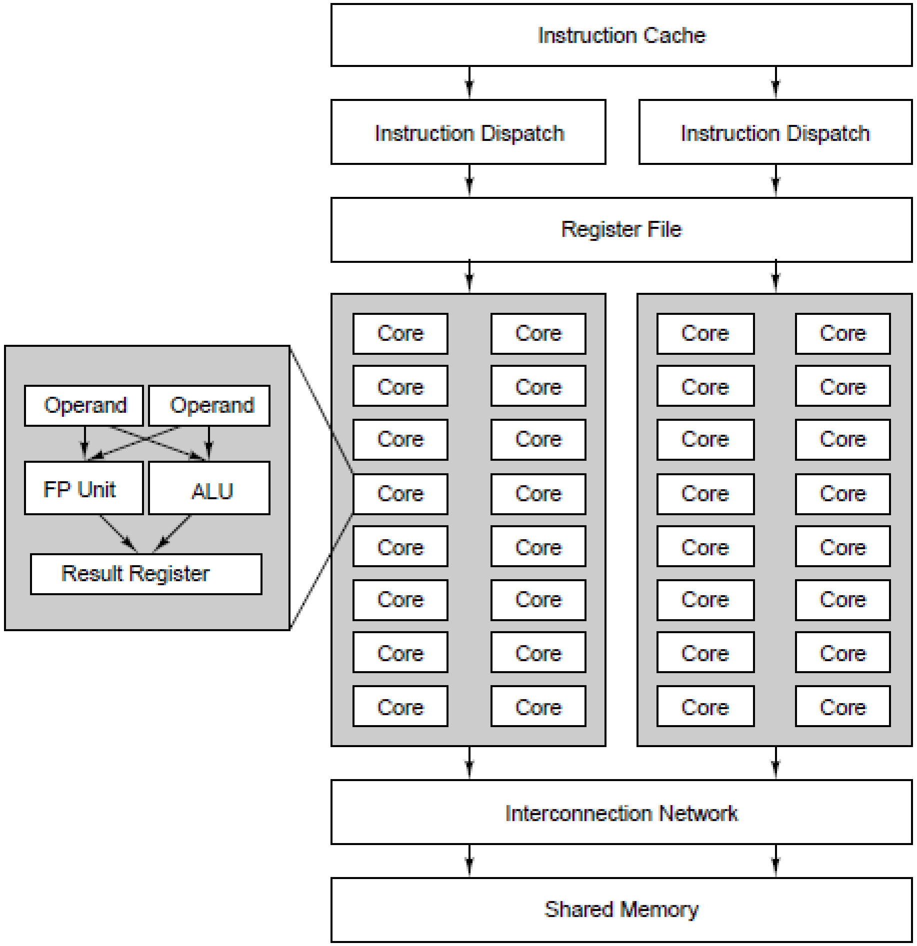


Superscalar Architectures

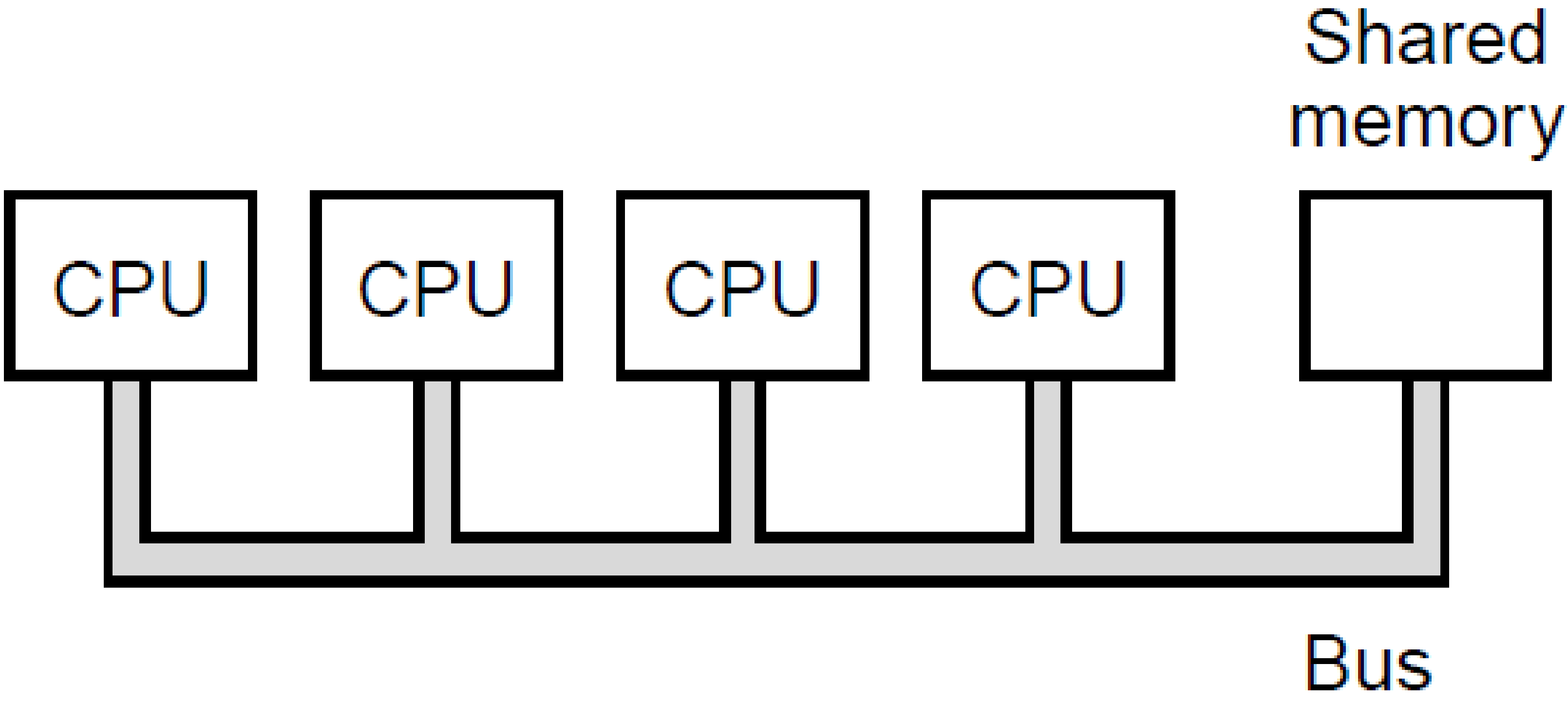
- Intuition: S3 stage issues instructions considerably faster than the S4 stage can execute them



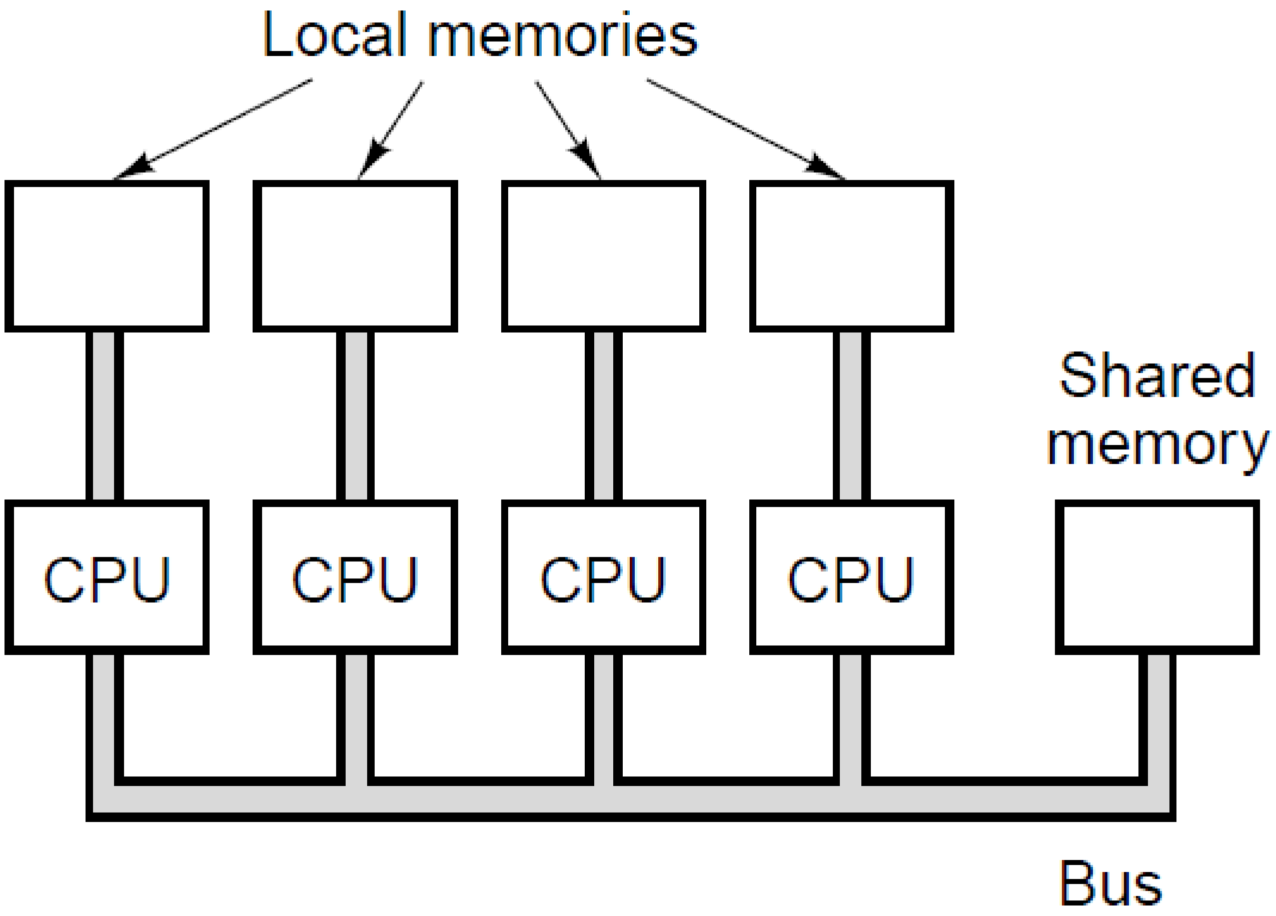
Data Parallel Computers



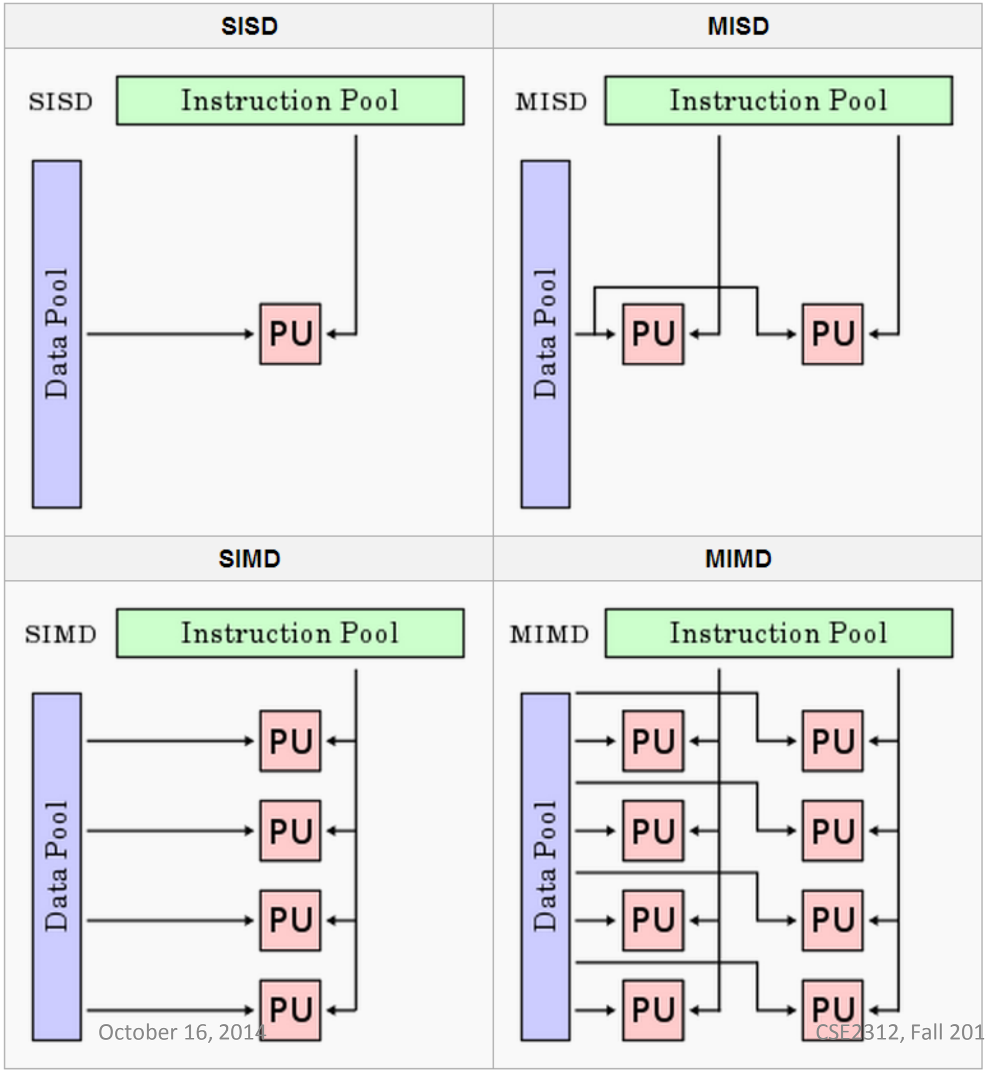
Multiprocessors



Multicomputers



Flynn's Taxonomy



- SISD: Single Instruction, Single Data
  - Classical Von Neumann
- SIMD: Single Instruction, Multiple Data
  - GPUs
- MISD: Multiple Instruction, Single Data
  - More exotic: fault-tolerant computers using task replication (Space Shuttle flight control computers)
- MIMD: Multiple Instruction, Multiple Data
  - Multiprocessors, multicomputers, server farms, clusters, …

Review: Example .gdbinit

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

- 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

Review: 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 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

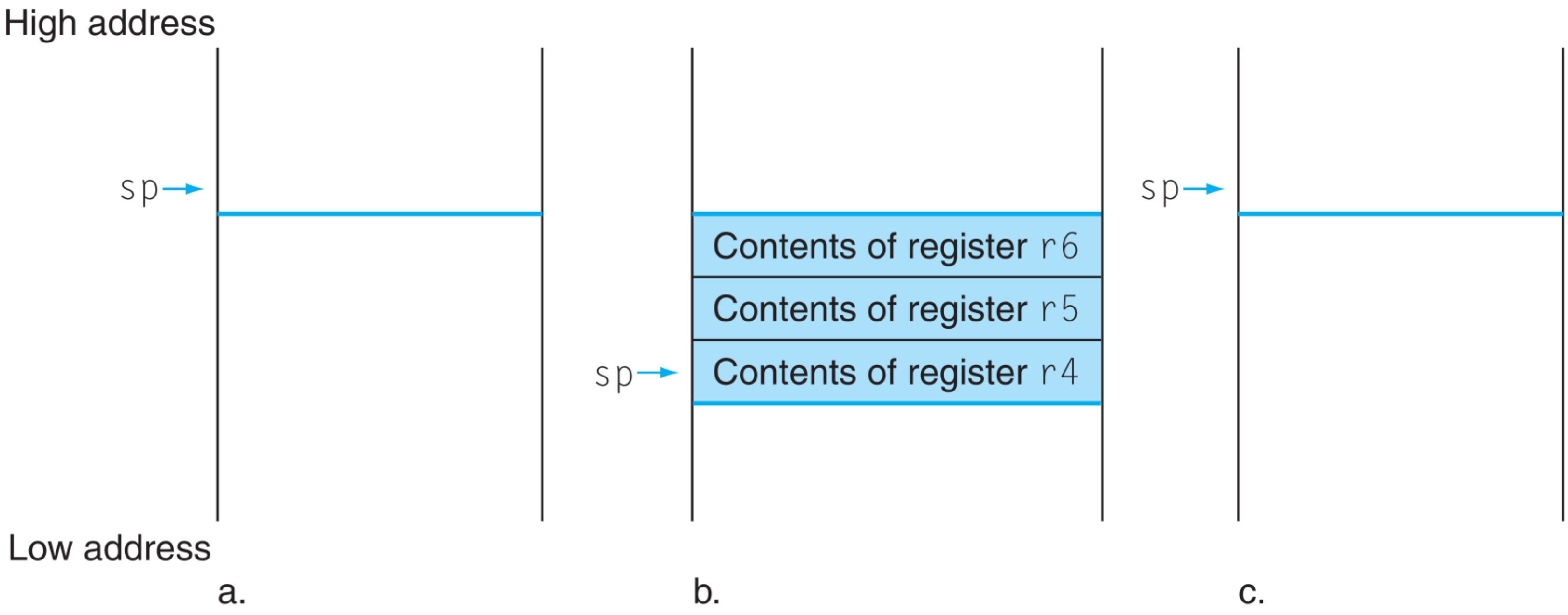


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

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

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

- What about strings (character arrays, i.e., multiple characters)?
- Strings are stored in memory at consecutive addresses
  - Like arrays that we saw last time

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)
```

Summary

- Pipelines
  - Instruction-level parallelism
    - Running pieces of several instructions simultaneously to make the most use of available fixed resources (think laundry)
  - Other forms of parallelism: Flynn's taxonomy

- Know what make does
- Know how to start QEMU
- Know how to start GDB
- Start learning how to interact and debug with GDB
  - Saw example of debugging the stack, etc.

Questions?

?