COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

What we have done so far

- Data processing/ALU instructions
- Memory instructions
- Branch instructions
- Various addressing modes
- How do we use these instructions to translate structured programs with loops into ARM assembly?
- ISA enables easier translation of high-level language constructs

Recall: Conditional Branch

- Conditional branch uses condition mnemonics

```
Assembly code:
    MOV  R0,  #4
    ADD  R1,  R0,  R0
    CMP  R0,  R1
    BEQ  THERE
    ORR  R1,  R1,  R1
THERE
    ADD  R1,  R1,  #78
```

- CMP subtracts R1 from R0 and **sets** all flags
  - Z flag is FALSE because R0 – R1 is not 0

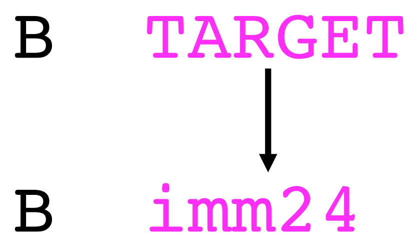
- The branch **BEQ** evaluates to FALSE
  - Branch is NOT TAKEN (NT)
  - The next instruction executed is the ORR instruction

Instruction Format – 3: Branch

31:28	27:26	25:24	23:0
cond	op	1L	imm24

- op = 10
- imm24 = 24-bit **signed** immediate
- The two bits [25:24] form the funct field
  - Bit 25 is always 1
  - **L bit:** L = 0 for **B** (Branch)
  - **L bit:** L = 1 for **BL** (Branch and Link)

- Format



B TARGET

B imm24

Branch with L = 0

- Branch with **L** bit (Bit 24) as 0 is a regular branch

31:28	27:26	25:24	23:0
cond	10	10	imm24

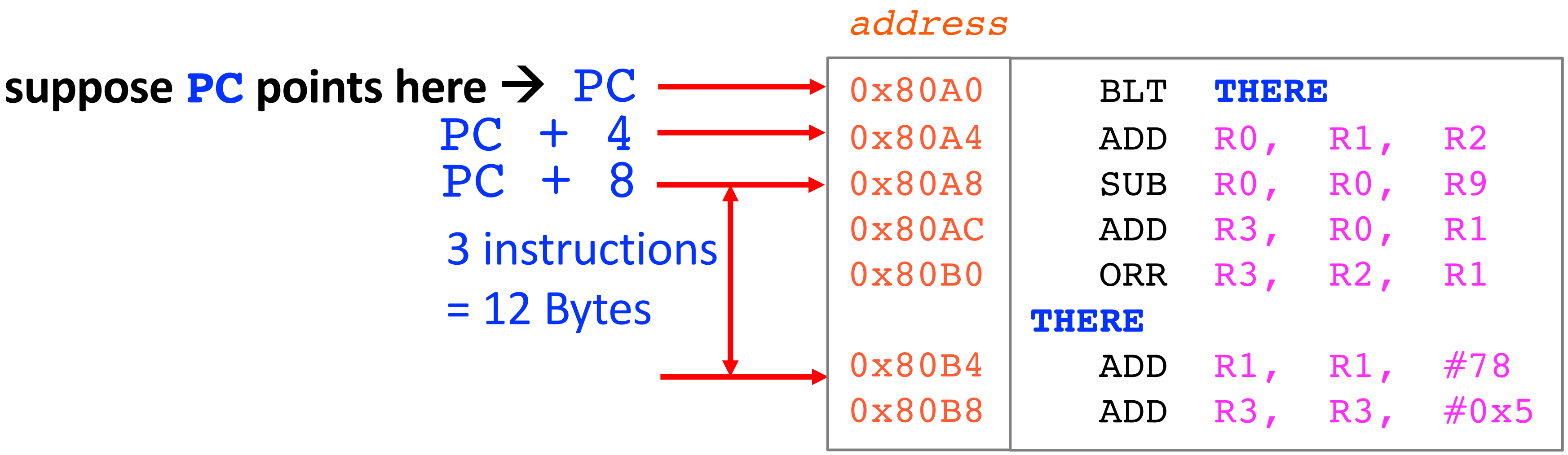
- **Branch Target Address (BTA)**: The address of the next instruction to execute if the branch is taken

- How is **BTA** calculated?
  1. Shift left `imm24` by 2 (to convert **words** to **bytes**)
  2. Sign-extend (copy `Instruction[23]` into `Instruction[24:31]`)
  3. Add `PC + 8`

BTA Calculation Example

- Instruction encodes the distance from `PC + 8` as three 32-bit words

31:28	27:26	25:24	23:0
cond	10	10	imm24 = 3 (000000000000000000000011)



suppose **PC** points here → PC

PC + 4

PC + 8

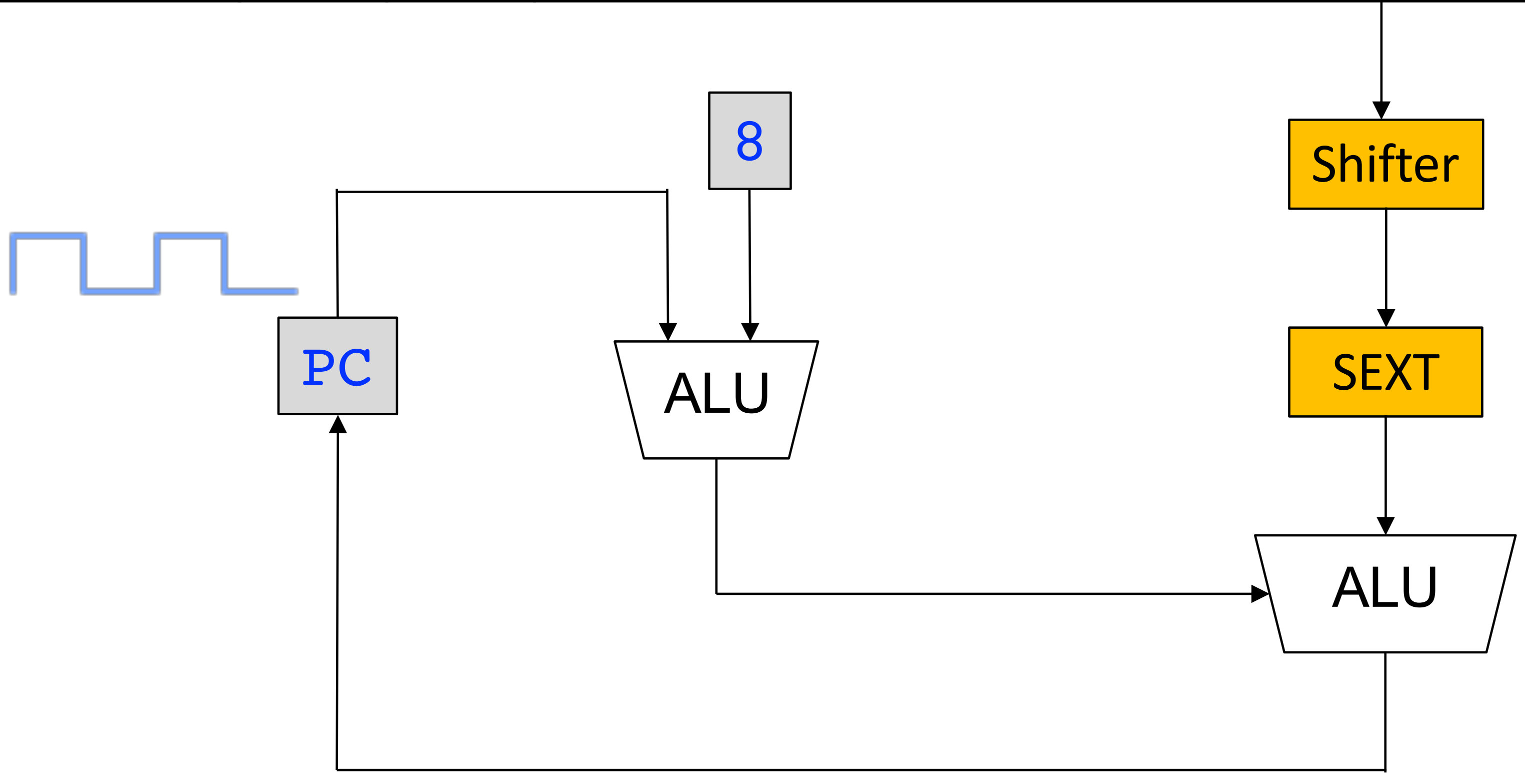
3 instructions

= 12 Bytes

```
address
0x80A0       BLT   THERE
0x80A4       ADD   R0,   R1,   R2
0x80A8       SUB   R0,   R0,   R9
0x80AC       ADD   R3,   R0,   R1
0x80B0       ORR   R3,   R2,   R1
           THERE
0x80B4       ADD   R1,   R1,   #78
0x80B8       ADD   R3,   R3,   #0x5
```

BTA Calculation DataPath

31:28	27:26	25:24	23:0
cond	10	10	imm24 = 3 (0000000000000000000000011)



BTA Calculation Summary

The processor calculates the **BTA** in three steps

1. Shift left imm24 by 2 (to convert words to bytes)
2. Sign-extend (copy Instr_{23} into $\text{Instr}_{31:24}$)
3. Add PC + 8

								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	= 3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	= 12

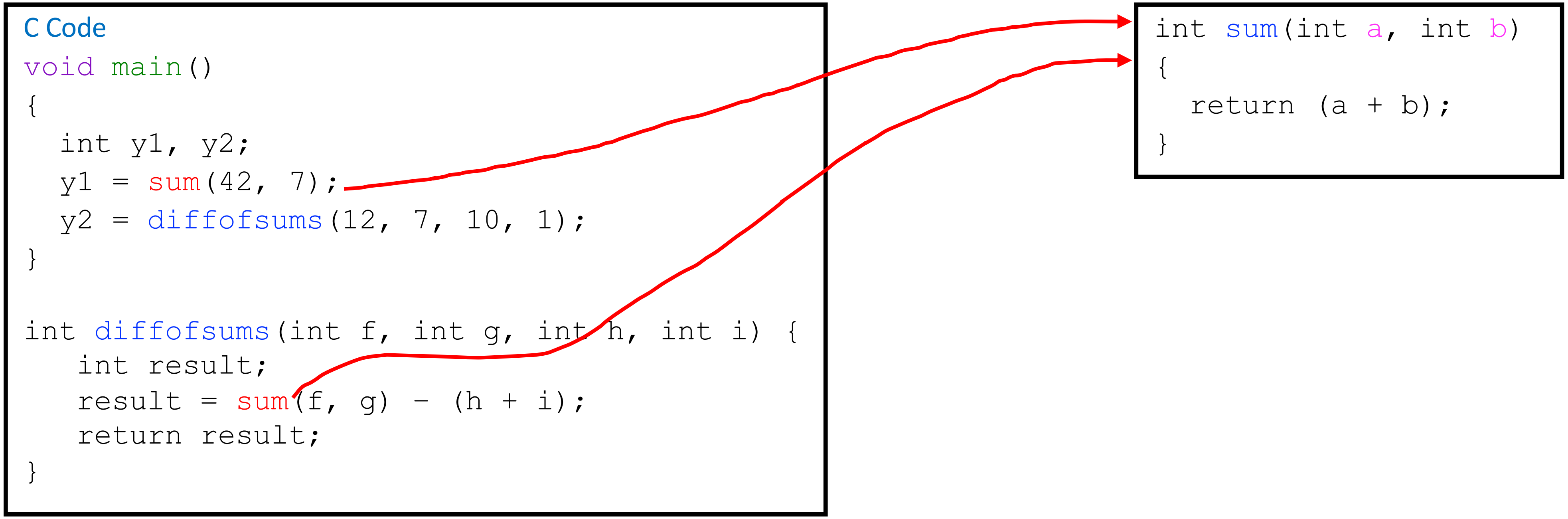
PC-Relative Addressing Mode

Add an offset to the PC to calculate the memory address of the target instruction

Functions

Functions

- Program fragments that are written once and invoked multiple times within the **same or a different** program



C Code

```
void main()
{
  int y1, y2;
  y1 = sum(42, 7);
  y2 = diffofsums(12, 7, 10, 1);
}

int diffofsums(int f, int g, int h, int i) {
   int result;
   result = sum(f, g) - (h + i);
   return result;
}
```

```
int sum(int a, int b)
{
  return (a + b);
}
```

- One software engineer writes `sum(int a, int b)` and many others can **reuse** it

Libraries of Pre-Existing Functions

- One might require a **fragment** that has been supplied by the manufacturer or by some independent software supplier

- It is almost always the case that collections of such fragments are available to programmers to **free** them from having to write their own

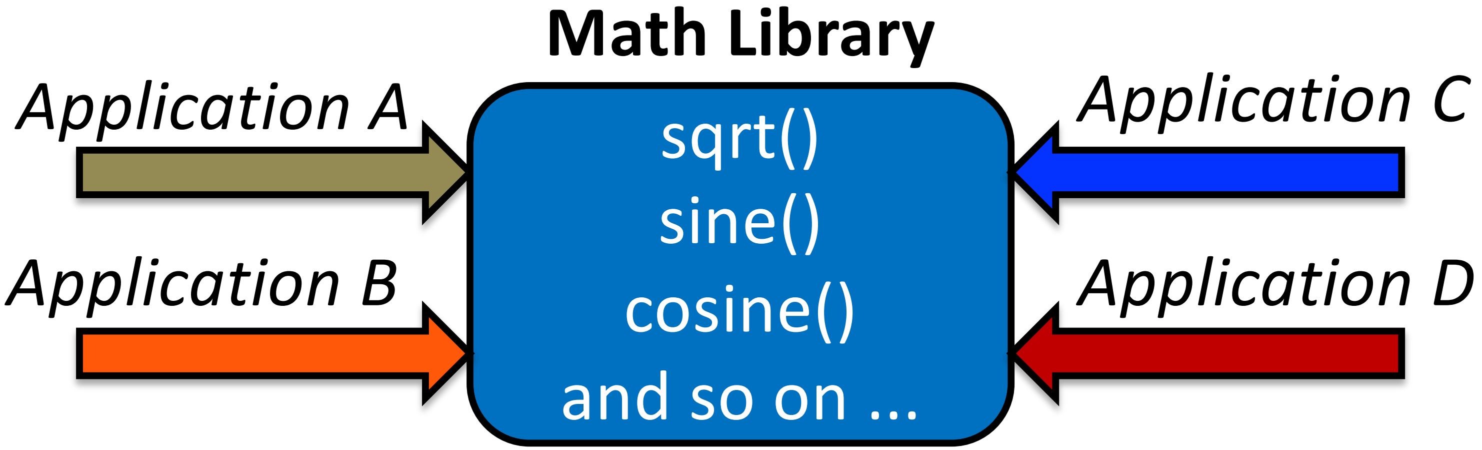
- These collections are called libraries

- Example: Math library provides square root, sine, cosine, arctangent

- Programmers do not need to reinvent the wheel

API

- Application **P**rogramming **I**nterface (**API**)
  - Defines the interfaces by which one software program communicates with another **at the source code level**



- **API defines the interface only**
  - The user of the API can ignore the implementation
  - Many implementations of the same API
  - C standard library hides many low-level details of the system

Usefulness of Functions

- High-level languages offer functions to enable
  - Abstraction & Modularity
  - Code reuse
  - Readability
  - Testability & validation
  - Maintainability

Functions

- Functions are also called **procedures** or **subroutines**

- Functions are ubiquitous which encourages ISA support
  - Special jump instructions
  - Special (isolated) space in memory to store function-related data
  - “Mechanism” to reduce interference between nested functions

What we will cover

- Architectural support for functions
  - Branch and Link instruction (BL)
  - Stack Pointer (SP)
  - Link Register (LR)

- Microarchitecture-level impact of programming styles
  - Iteration vs. Recursion

- Get a deeper understanding of hardware/software interaction and tradeoffs

Functions in C

- `main()` is the caller
  - It calls another function
  - Returns nothing (void)
  - Takes no input arguments
- `sum()` is the callee
  - Being called by some function
  - Takes two input arguments of type int: a and b
  - It returns an integer value
  - Computes the sum of a and b

C Code

```
void main()
{
   int y;
   y = sum(42, 7);
   ...
}

int sum(int a, int b)
{
   return (a + b);
}
```

Functions in C

- The caller provides the input arguments
  - 42 and 7 in this example

C Code

```c
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

- The distinction between caller and callee depends on the context
  - What if someone calls the `main` function?

Leaf versus Non-Leaf Functions

- `sum()` is a **leaf** function
  - It does not call another function

- `main()` is a non-leaf function
  - It calls another function

- Non-leaf functions are **more complicated** especially at the assembly level

Functions as Detectives

- Assigned a secret mission (function call)

- Acquires necessary resources (acquire memory)

- Perform the mission (execute instructions)

- Leaves no trace (clean up memory)

- Returns safely to point of origin (function return)

Breaking Down Function Execution

- Caller **stores** arguments in registers or memory
- Function call: Caller **transfers** flow control to the callee
- Callee **acquires/allocates** memory for doing work
- Callee **executes** the function body
- Callee **stores** the result in “some” register
- Callee **deallocates** memory
- Function return: Callee **returns** control back to the caller

Instruction for Function Call

- It is usually the case that ISAs provide a special variant of the branch instruction for making the function call
  - MIPS : **jal**
  - ARM : **BL**
  - Intel x86 : **call**
  - RISC-V : **jal**
  - QuAC: No architectural support for functions

ARM Function Call

- BL (Branch and Link)
  - CPU branches to the label specified by BL
  - CPU stores the **return address** in the link register (LR)
  - Return address is the address of the next instruction after the function call
  - **How should we return from the function to the caller?**

Returning to Callee

- Returning from function requires updating the PC
  - Move the link register into `PC`
  - `MOV PC, LR`
- How should we pass arguments to the function?
- Where should we return the value?

Passing Arguments

- Passing arguments (convention)
  - `R0, R1, R2, R3`

- Returning value (convention)
  - `R0`

ARM Register Set

Name	Use
`R0`	Argument / return value / temporary variable
`R1-R3`	Argument / temporary variables
`R4-R11`	Saved variables
`R12`	Temporary variable
`R13 (SP)`	Stack Pointer
`R14 (LR)`	Link Register
`R15 (PC)`	Program Counter

Example of a Function Call

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

Memory Address	ARM Assembly Code
0x00000200	MAIN **BL SIMPLE**
0x00000204	ADD R4, R5, R6
...	
0x00401020	SIMPLE **MOV PC, LR**

- **BL**   branches to SIMPLE
  LR = PC + 4 = **0x00000204**
- **MOV PC, LR**   sets PC = LR
  (the next instruction executed is at **0x00000204**)

- MAIN and SIMPLE are labels (memory addresses) in assembly
- BL transfers flow to SIMPLE and stores the *return address* in LR
- The function returns after MOV, and the next instruction (ADD) is executed

Another Example: Difference of Sums

```
C code:
int main() {
   int y;
   ...
   y = diffofsums(2, 3, 4, 5);
   ...
}

int diffofsums(int f, int g, int h, int i) {
   int result;
   result = (f + g) – (h + i);
   return result;
}
```

ARM Assembly Code

```
; R4 = y

MAIN
   ...
   MOV R0, #2          ; argument 0 = 2
   MOV R1, #3          ; argument 1 = 3
   MOV R2, #4          ; argument 2 = 4
   MOV R3, #5          ; argument 3 = 5
   BL  DIFFOFSUMS      ; call function
   MOV R4, R0          ; y = returned value
   ...

; R4 = result
DIFFOFSUMS
   ADD R8, R0, R1      ; R8 = f + g
   ADD R9, R2, R3      ; R9 = h + i
   SUB R4, R8, R9      ; result = (f + g) - (h + i)
   MOV R0, R4          ; put return value in R0
   MOV PC, LR          ; return to caller
```

Questions

- How can we pass more than 4 function arguments?

- How can we ensure that registers in use by the caller are not corrupted?
  - `DIFFOFSUMS` overwrites `R4, R8, R9`
  - `MAIN` may have **"live"** values in these registers

- Answer: **The Stack**
  - A special area in memory used across function calls to preserve registers and store temporary values that overflow available registers

Aside: Register Reuse

- **Live register**
  - A subsequent instruction will use the register value
  - The property is called **liveness**
- **Dead register**
  - No upcoming instruction will use the register’s value
  - The register can be safely used to store a new value
  - It’s really the value stored in the register that is dead and not the physical register

Uses of Stack

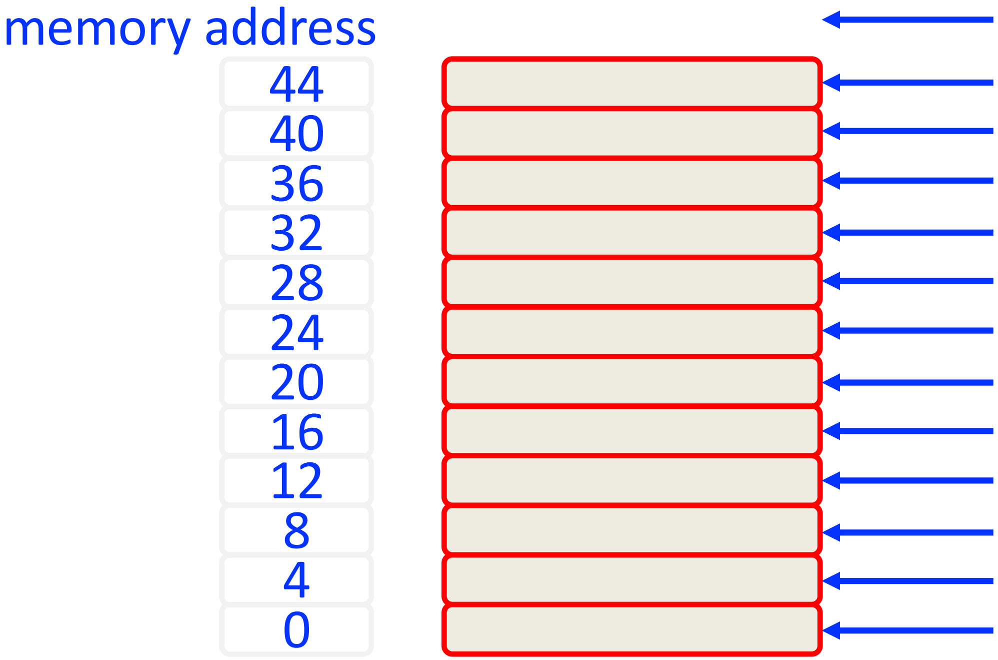
- Preserving and saving registers

- Passing extra arguments

- Temporary memory space for Storing **function-local** variables

The Stack

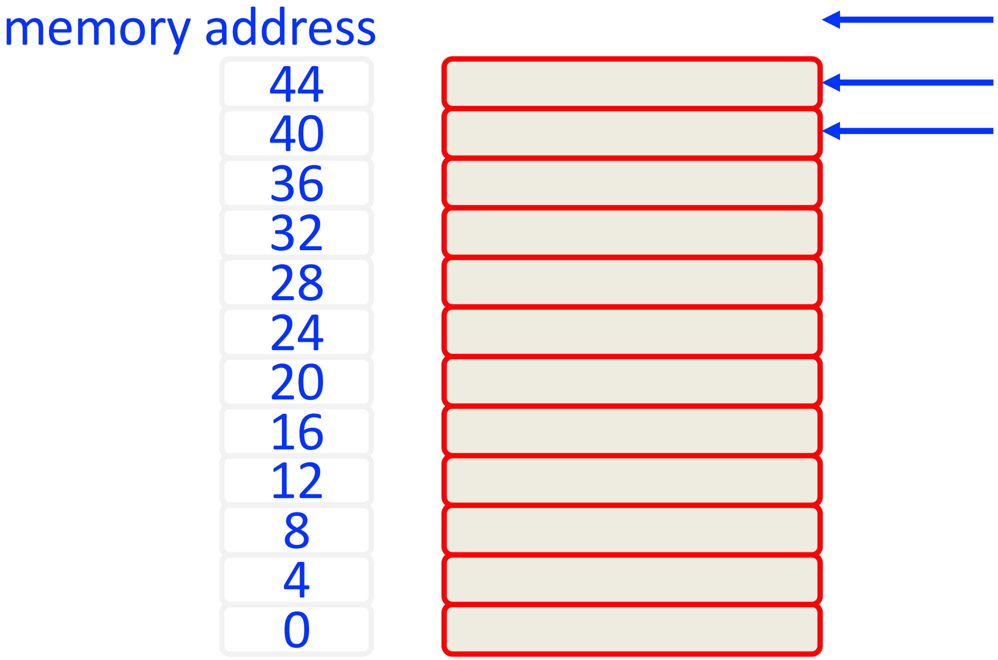
- A stack is like a Last In First Out (LIFO) Queue



- Stack **expands** and **contracts** as items are added and removed

The Stack

- A stack is like a Last In First Out (LIFO) Queue



- Stack **expands** and **contracts** as items are added and removed

Implementing a Stack

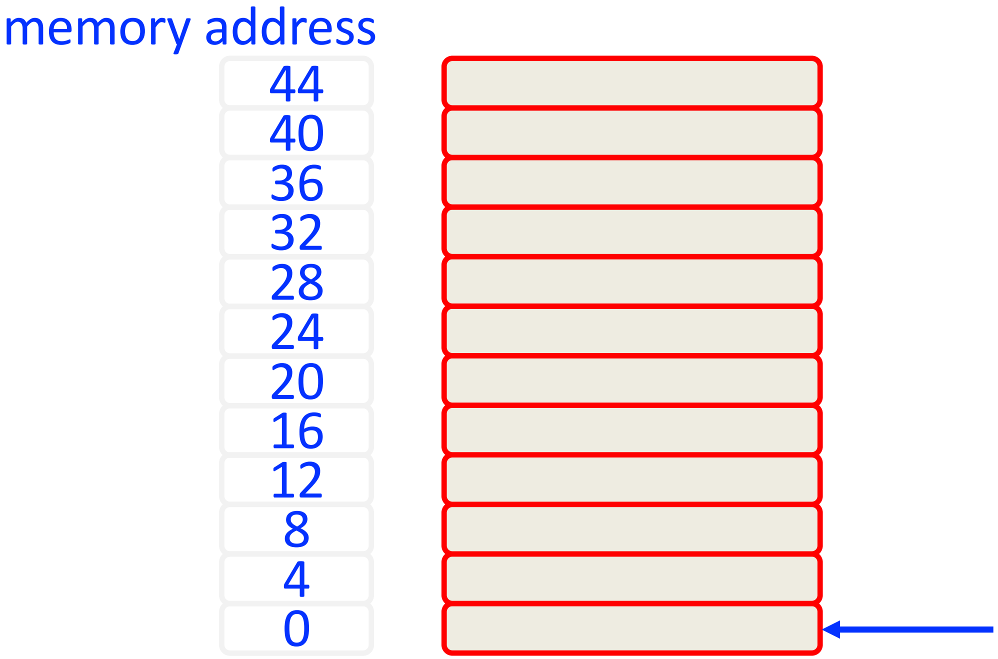
- What do we need to implement a stack?

- We need “some memory” for stack data (items)
  - Do we have memory? Yes, we can use data memory

- We need “an arrow” to point to the top of the stack
  - What does an arrow represent in comp. architecture?
  - It represents a pointer to a memory location
  - In other words, a register containing the memory address
  - Do we have a register? Yes, we can use an architectural register
  - **Stack Pointer (SP):** An architectural register that is **by convention** dedicated to storing the top of the stack

Implementing a Stack

- Suppose we have the stack pointer initialized to 0. How do we make space for (aka reserve) **8 items** on the stack. Word size = 4 bytes



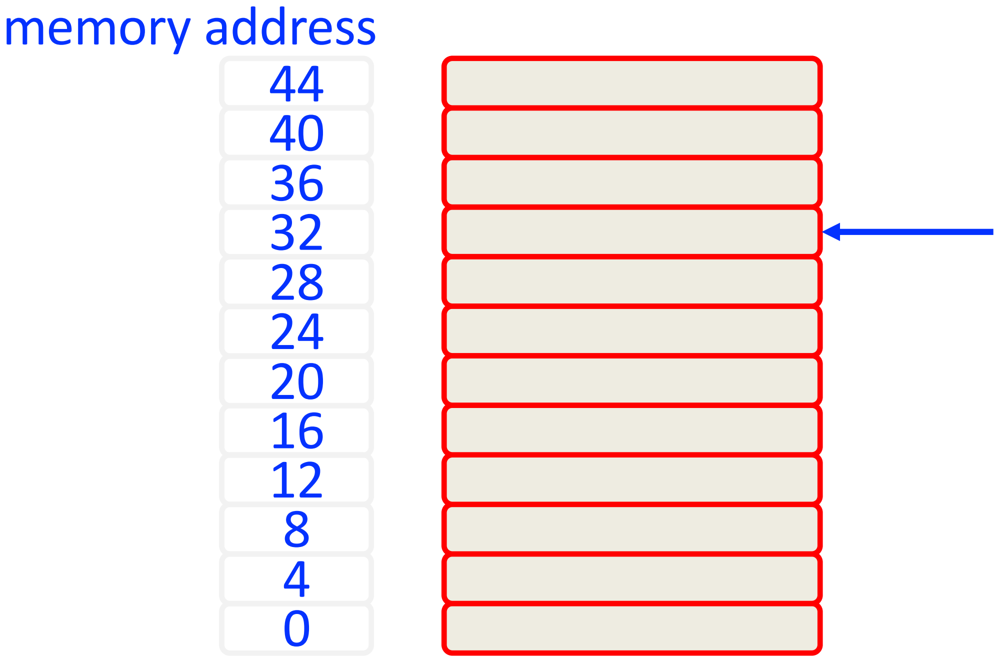
memory address

44
40
36
32
28
24
20
16
12
8
4
0

- Add 32 to the stack pointer: SP = SP + 32

Implementing a Stack

- Suppose we have the stack pointer initialized to 0. How do we make space for (aka reserve) **8 items** on the stack. Word size = 4 bytes



- Add 32 to the stack pointer: SP = SP + 32

Growing and Shrinking the Stack

- `push`
  - Put a new item on top of the stack and adjust the stack pointer accordingly (`SP = SP + 4`)

- `pop`
  - Remove an item from top of the stack and subtract 4 from the stack pointer

Push and Pop Operations

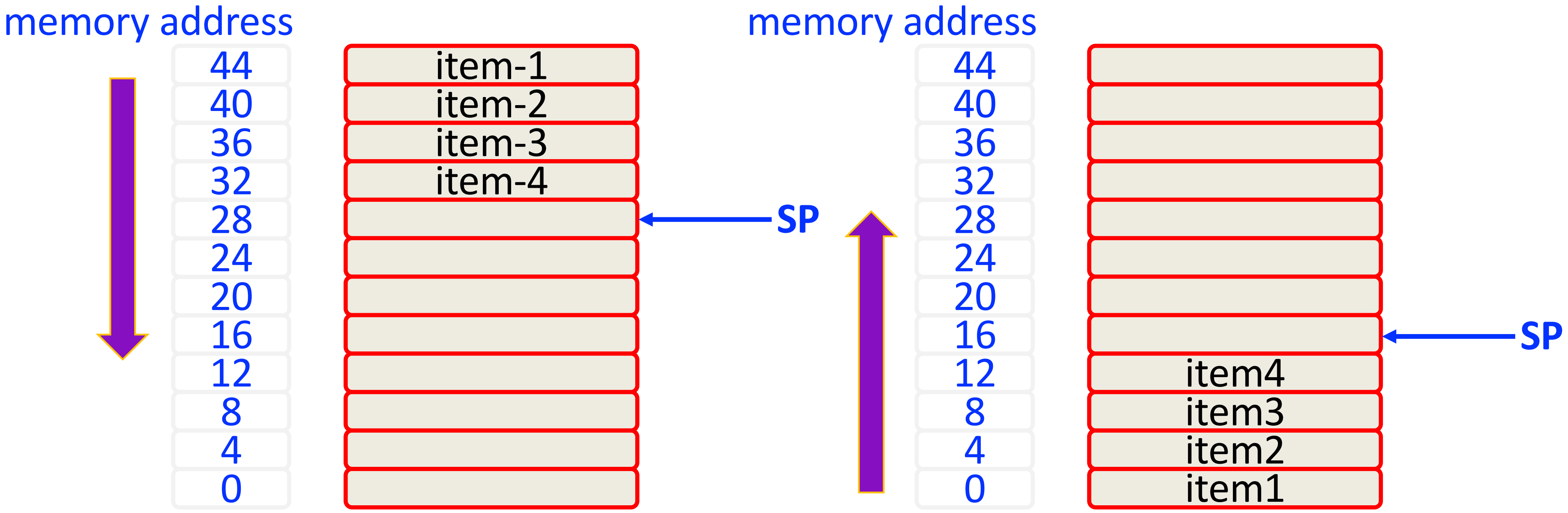
- We store the stack at “some” arbitrary address in memory
  - Details of how the address is chosen are not important

- `push {R0}`
  - Store `R0` onto the stack

- `pop {R0}`
  - Restore `R0` with whatever is at the top of the stack

Different Ways to Manage Stack

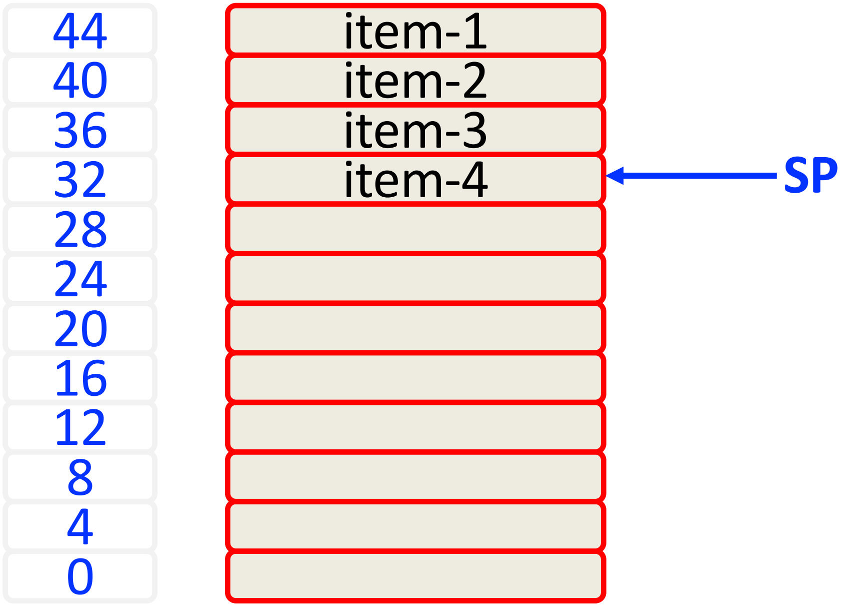
- **Descending** Stack
  - Grows downward
  - SP points to the lowest address
- **Ascending** Stack
  - Grows upward
  - SP points to highest address



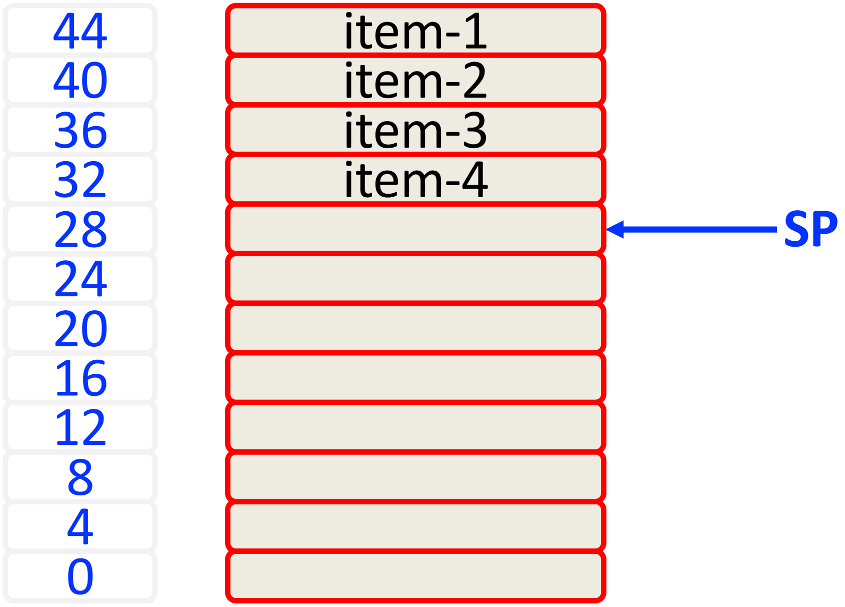
Further Classification

- **Full** Stack
  - SP **points to** the last allocated space on the stack (top)
  - Last item pushed
- **Empty** Stack
  - SP **points to one word beyond the top of stack**

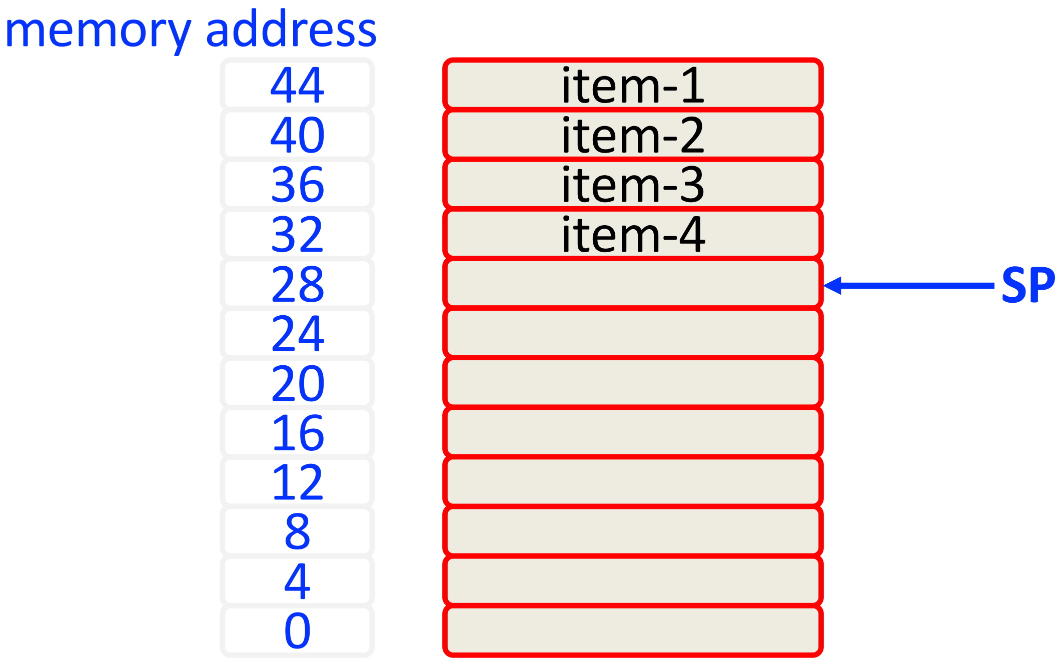
memory address



memory address

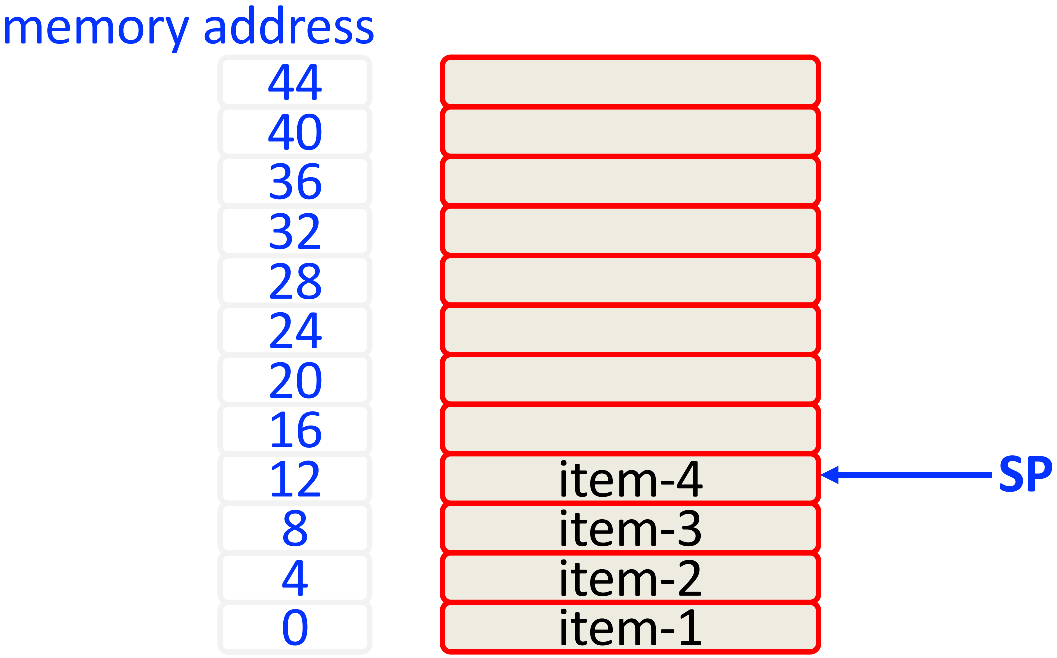


Empty Descending Stack



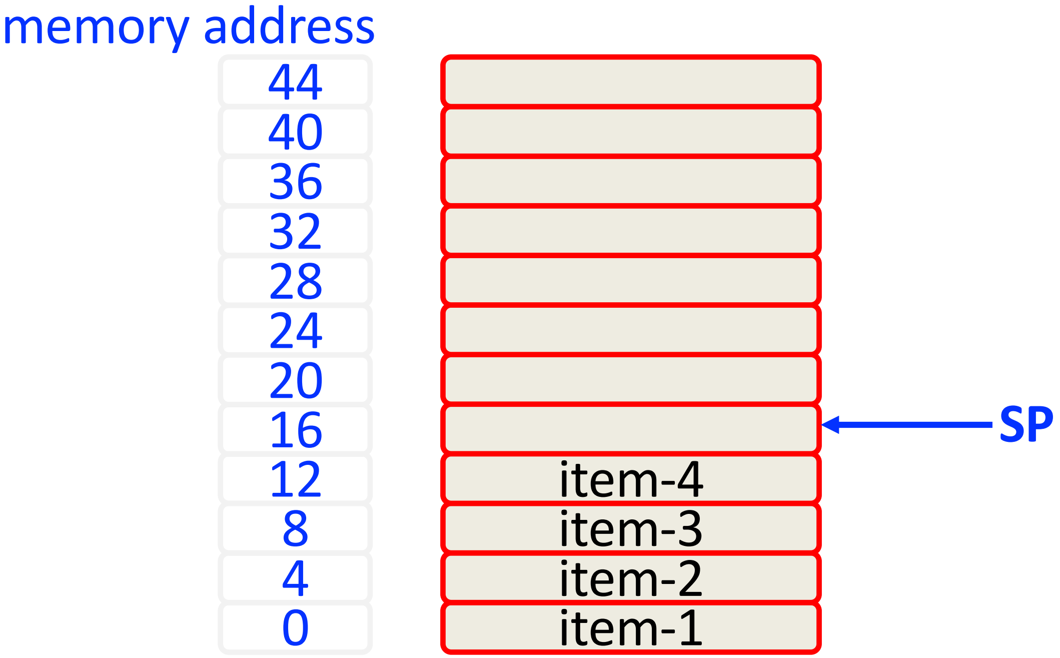
memory address		
44	item-1	
40	item-2	
36	item-3	
32	item-4	
28		← SP
24		
20		
16		
12		
8		
4		
0		

Full Ascending Stack



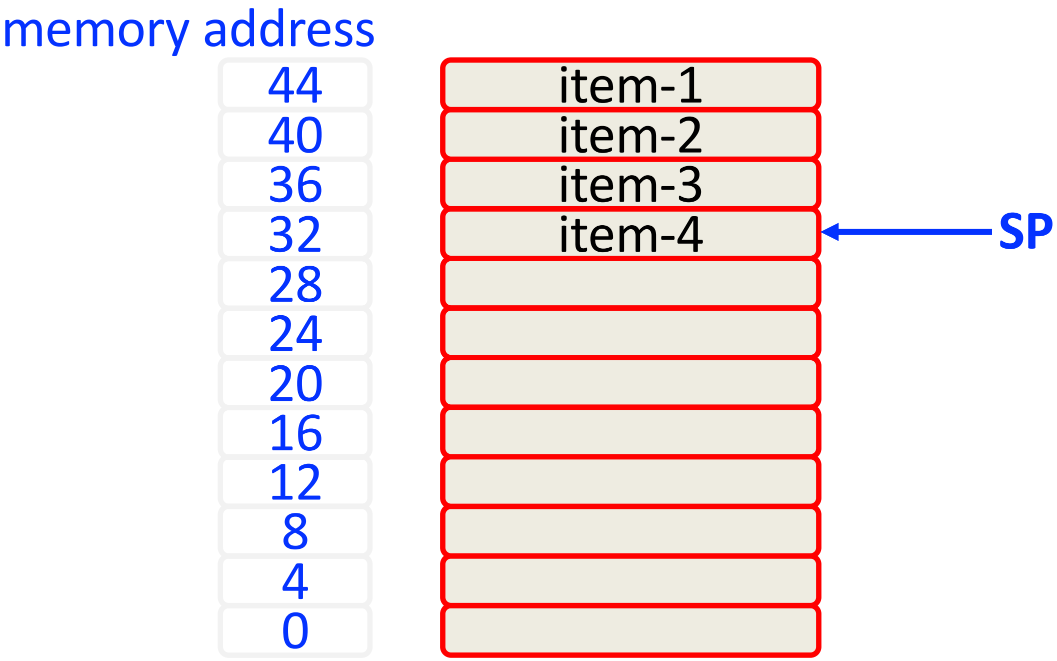
memory address		
44		
40		
36		
32		
28		
24		
20		
16		
12	item-4	← SP
8	item-3	
4	item-2	
0	item-1	

Empty Ascending Stack



memory address		
44		
40		
36		
32		
28		
24		
20		
16		SP
12	item-4	
8	item-3	
4	item-2	
0	item-1	

Full Descending Stack



memory address		
44	item-1	
40	item-2	
36	item-3	
32	item-4	SP
28		
24		
20		
16		
12		
8		
4		
0		

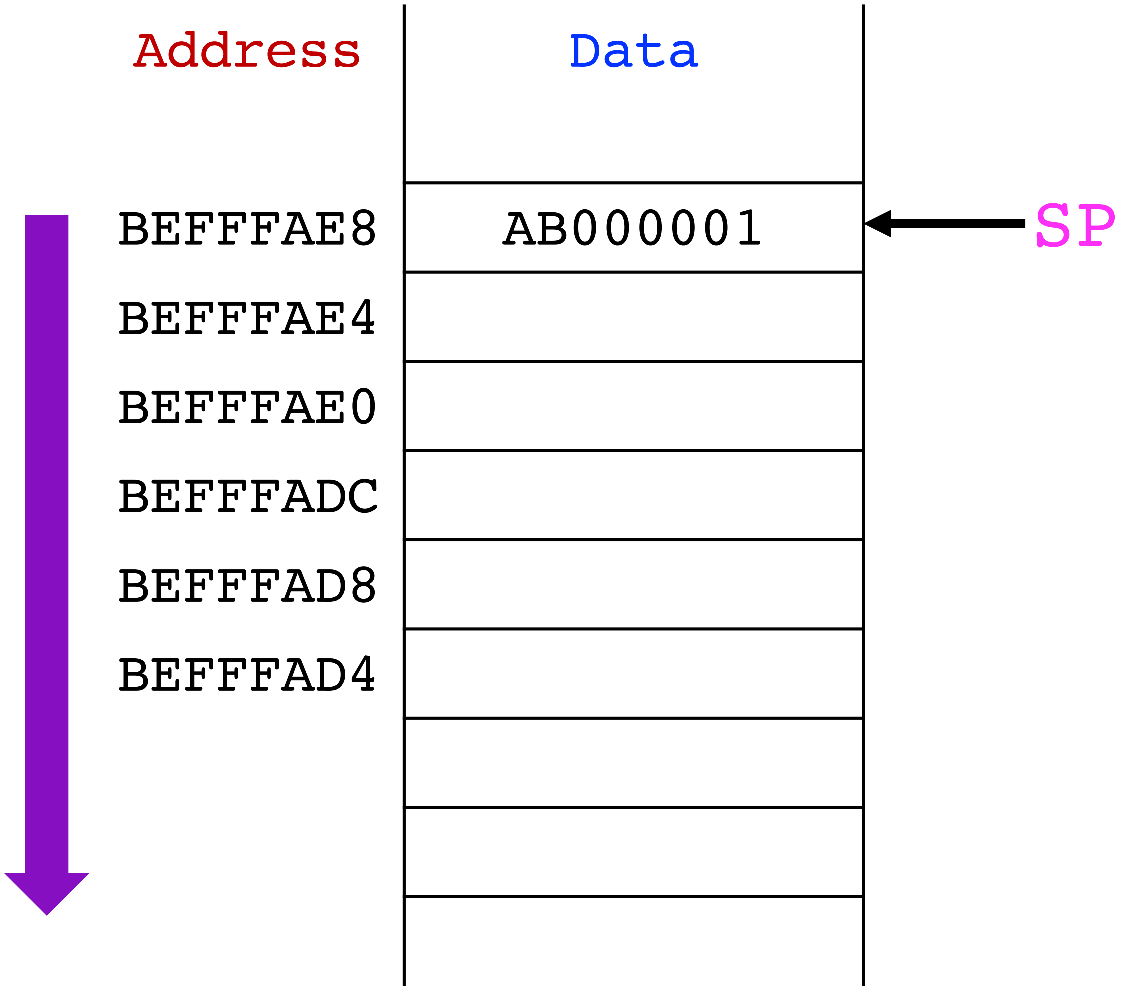
- ARM specifies a full descending stack, which we will assume in this course

ARM Stack

- ARM stack **grows down** in memory
- Stack Pointer (SP) points to the **top of the stack**
  - SP register holds the address of (points to) the **top of the stack**

contents of stack pointer

SP `0xBEFFFAE8`



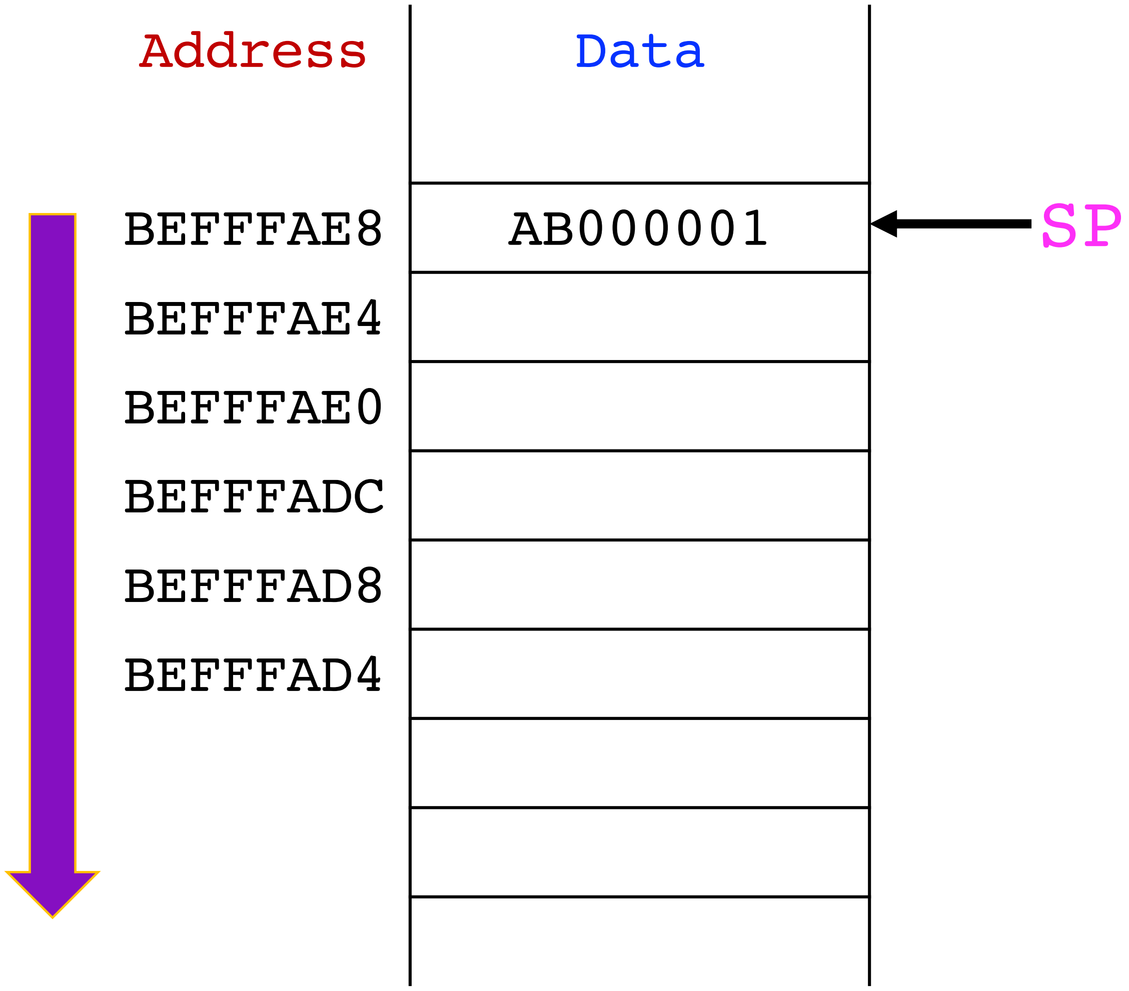
Address	Data
BEFFFAE8	AB000001
BEFFFAE4	
BEFFFAE0	
BEFFFADC	
BEFFFAD8	
BEFFFAD4	

Growing the Stack

- Let us push two items on the stack
  - 0x12345678
  - 0xFFFFDDCC
- Where does the SP points to now?
- How does the stack look?

contents of stack pointer

SP 0xBEFFFAE8



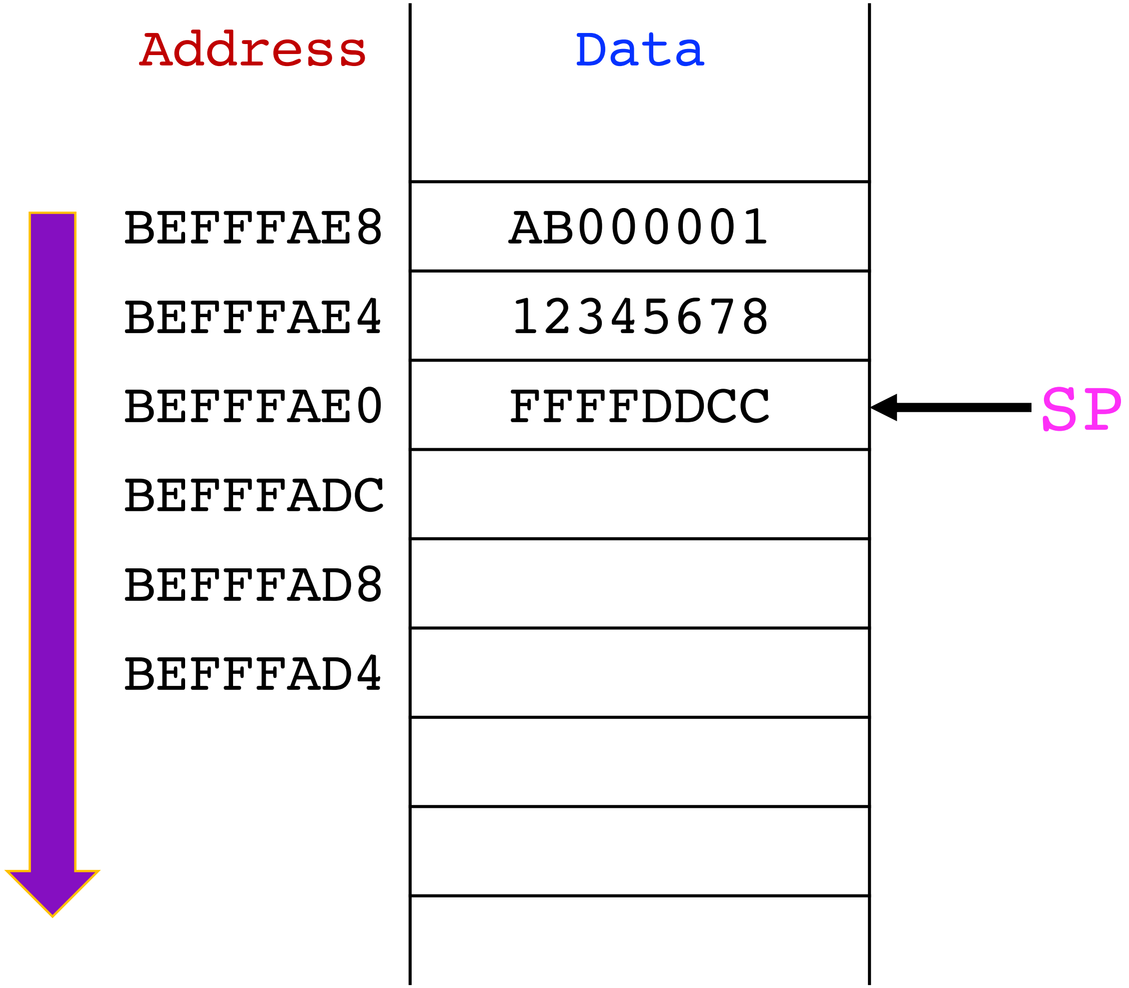
Address	Data	
BEFFFAE8	AB000001	← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		

Growing the Stack

- SP points to the most recently pushed item on the stack
- SP decrements by 8 to make space for two words

contents of stack pointer

SP 0xBEFFFAE0



Address	Data	
BEFFFAE8	AB000001	
BEFFFAE4	12345678	
BEFFFAE0	FFFFDDCC	SP
BEFFFADC		
BEFFFAD8		
BEFFFAD4		

Saving and Restoring Registers

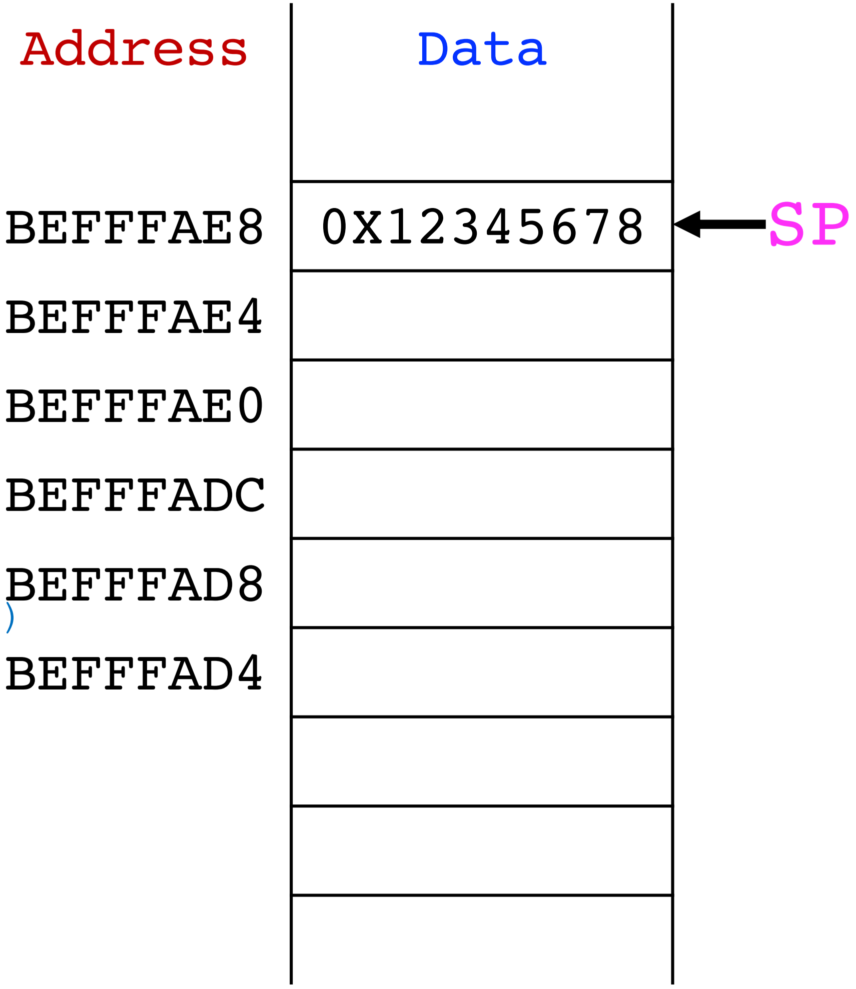
- `DIFFOFSUMS` **corrupts** three registers
  - Recall: Spy must not reveal their actions
  - No unintended side-effects (except leaving result in `R0`)
  - Callee should not corrupt caller’s execution

Saving and Restoring Registers

- Functions use the stack for saving/restoring registers
  - Allocate space on the stack (`SP = SP - 12`)
  - Store registers in use by the caller on the stack
  - Execute the function
  - Restore the registers from the stack
  - Deallocate space on the stack (`SP = SP + 12`)

Improved DIFFOFSUMS

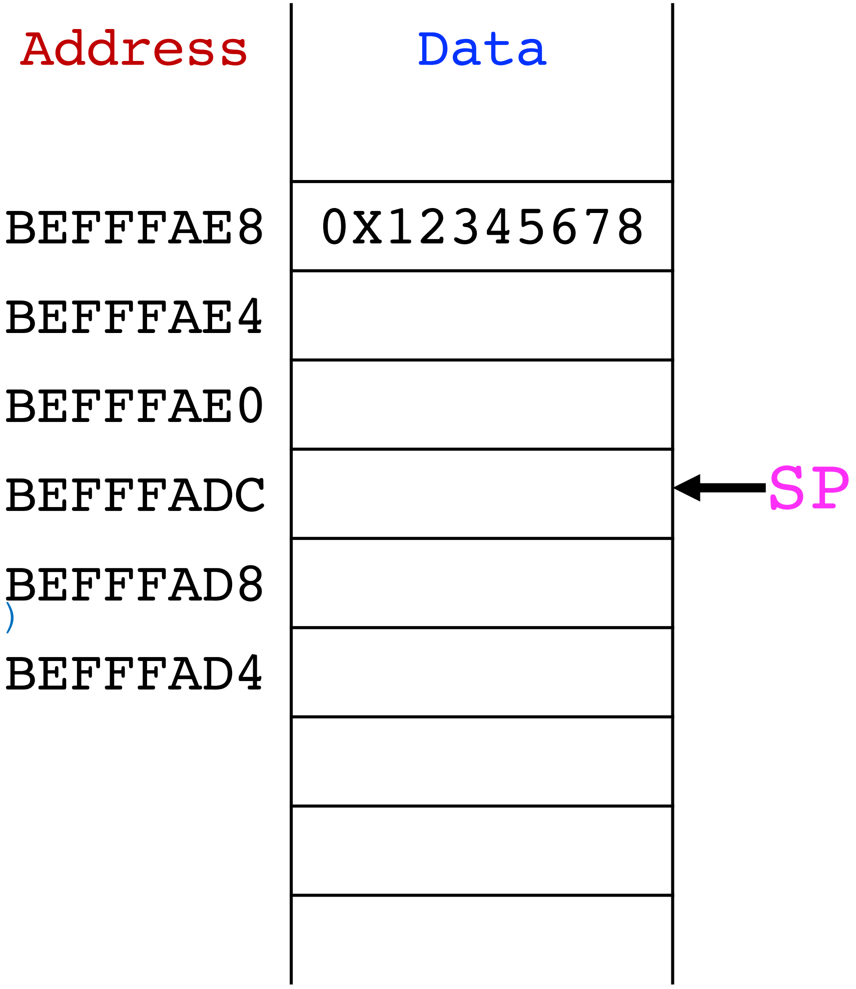
```
ARM Assembly Code
; R0 = result
DIFFOFSUMS
  SUB SP, SP, #12      ; make space on stack
                       ; for 3 registers
  STR R9, [SP, #8]     ; save R9 on stack
  STR R8, [SP, #4]     ; save R8 on stack
  STR R4, [SP]         ; save R4 on stack
  ADD R8, R0, R1       ; R8 = f + g
  ADD R9, R2, R3       ; R9 = h + i
  SUB R4, R8, R9       ; result = (f + g) - (h + i)
  MOV R0, R4           ; put return value in R0
  LDR R4, [SP]         ; restore R4 from stack
  LDR R8, [SP, #4]     ; restore R8 from stack
  LDR R9, [SP, #8]     ; restore R9 from stack
  ADD SP, SP, #12      ; deallocate stack space
  MOV PC, LR           ; return to caller
```



Address	Data	
BEFFFAE8	0X12345678	← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		

Improved DIFFOFSUMS

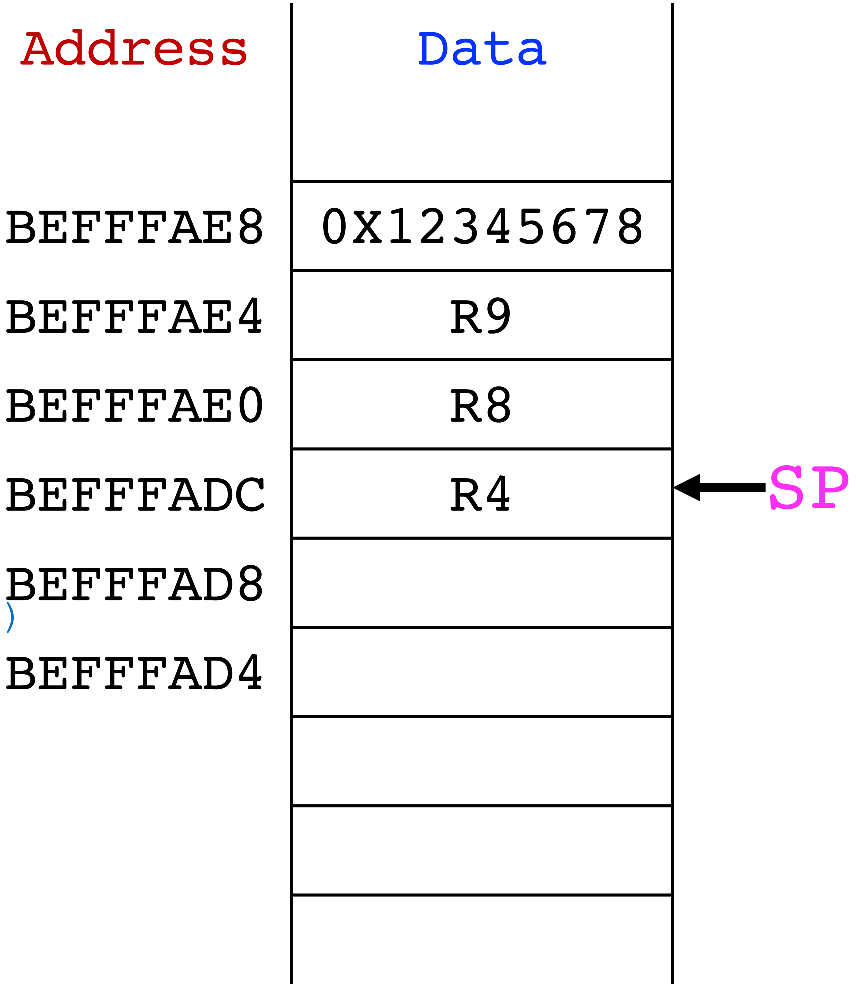
```
ARM Assembly Code
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12      ; make space on stack
                       ; for 3 registers
  STR R9, [SP, #8]     ; save R9 on stack
  STR R8, [SP, #4]     ; save R8 on stack
  STR R4, [SP]         ; save R4 on stack
  ADD R8, R0, R1       ; R8 = f + g
  ADD R9, R2, R3       ; R9 = h + i
  SUB R4, R8, R9       ; result = (f + g) - (h + i)
  MOV R0, R4           ; put return value in R0
  LDR R4, [SP]         ; restore R4 from stack
  LDR R8, [SP, #4]     ; restore R8 from stack
  LDR R9, [SP, #8]     ; restore R9 from stack
  ADD SP, SP, #12      ; deallocate stack space
  MOV PC, LR           ; return to caller
```



Address	Data	
BEFFFAE8	0X12345678	
BEFFFAE4		
BEFFFAE0		
BEFFFADC		← SP
BEFFFAD8		
BEFFFAD4		

Improved DIFFOFSUMS

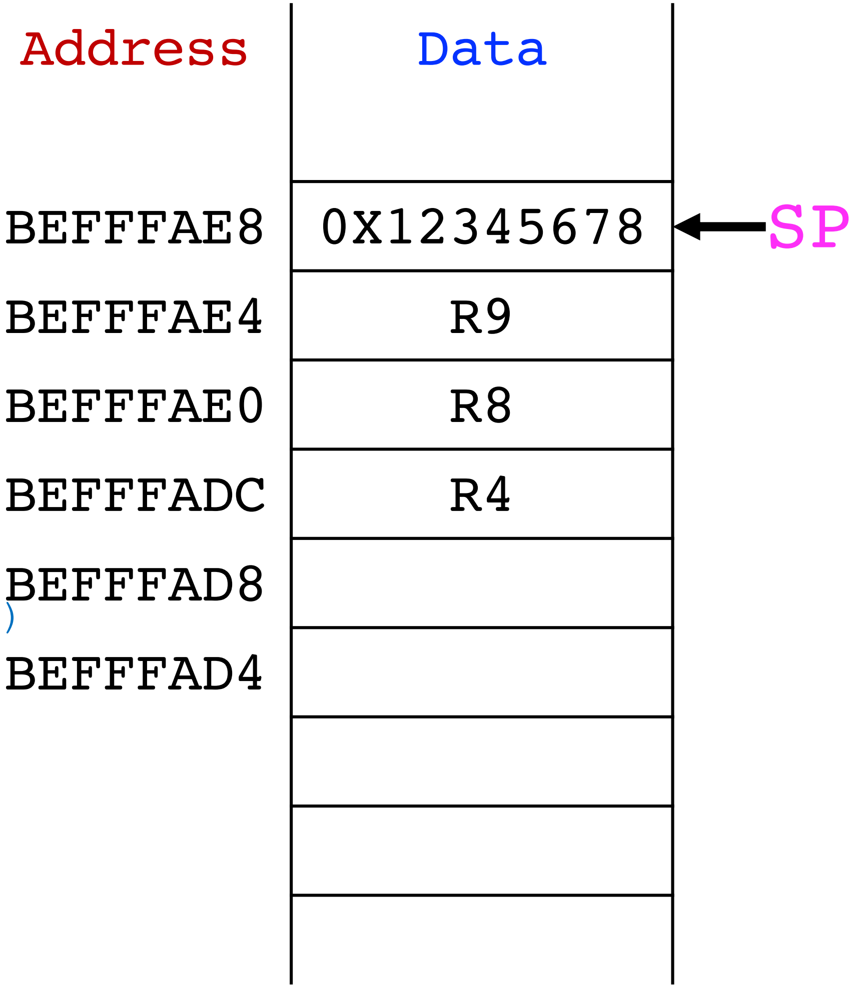
```
ARM Assembly Code
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12      ; make space on stack
                       ; for 3 registers
  STR R9, [SP, #8]     ; save R9 on stack
  STR R8, [SP, #4]     ; save R8 on stack
  STR R4, [SP]         ; save R4 on stack
  ADD R8, R0, R1       ; R8 = f + g
  ADD R9, R2, R3       ; R9 = h + i
  SUB R4, R8, R9       ; result = (f + g) - (h + i)
  MOV R0, R4           ; put return value in R0
  LDR R4, [SP]         ; restore R4 from stack
  LDR R8, [SP, #4]     ; restore R8 from stack
  LDR R9, [SP, #8]     ; restore R9 from stack
  ADD SP, SP, #12      ; deallocate stack space
  MOV PC, LR           ; return to caller
```



Address	Data
BEFFFAE8	0X12345678
BEFFFAE4	R9
BEFFFAE0	R8
BEFFFADC	R4
BEFFFAD8	
BEFFFAD4	

Improved DIFFOFSUMS

```
ARM Assembly Code
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12        ; make space on stack
                         ; for 3 registers
  STR R9, [SP, #8]       ; save R9 on stack
  STR R8, [SP, #4]       ; save R8 on stack
  STR R4, [SP]           ; save R4 on stack
  ADD R8, R0, R1         ; R8 = f + g
  ADD R9, R2, R3         ; R9 = h + i
  SUB R4, R8, R9         ; result = (f + g) - (h + i)
  MOV R0, R4             ; put return value in R0
  LDR R4, [SP]           ; restore R4 from stack
  LDR R8, [SP, #4]       ; restore R8 from stack
  LDR R9, [SP, #8]       ; restore R9 from stack
  ADD SP, SP, #12        ; deallocate stack space
  MOV PC, LR             ; return to caller
```



Address	Data	
BEFFFAE8	0X12345678	← SP
BEFFFAE4	R9	
BEFFFAE0	R8	
BEFFFADC	R4	
BEFFFAD8		
BEFFFAD4		

Calling Convention

- Preserving every register that a function uses is wasteful
  - `DIFFOFSUMS` preserves `R4, R8, R9`, but the caller may not be using `R8` or `R9`

- Calling convention is a contract that callers and callees must follow

Calling Convention

- With a convention in place
  - Functions written by different programmers can interoperate
  - Functions compiled by two different compilers can interoperate
  - A library function written by third party can safely be used without worrying about corruption due to misplaced arguments and return value

ARM Calling Convention

- Preserved Registers
  - Registers that are preserved across function calls
  - Caller can expect these registers to appear as if a function call was never made
  - **Callee must save and restore preserved registers**

- Nonpreserved Registers
  - Caller must save these registers before making the function call
  - **Their preservation is NOT the callee's responsibility**

ARM Calling Convention

Preserved	Nonpreserved
Saved registers: `R4 – R11`	Temporary register: `R12`
Stack pointer: `SP (R13)`	Argument registers: `R0 – R3`
Return address: `LR (R14)`	Current Program Status Register
Stack above the stack pointer	Stack below the stack pointer

- `SP` and `LR` are fancy names for `R13` and `R14`
- Stack above the stack pointer is preserved if the callee does not mess with the caller’s stack space (a.k.a. stack frame)
- Stack pointer is preserved, because the caller deallocates the space it uses on the stack before returning

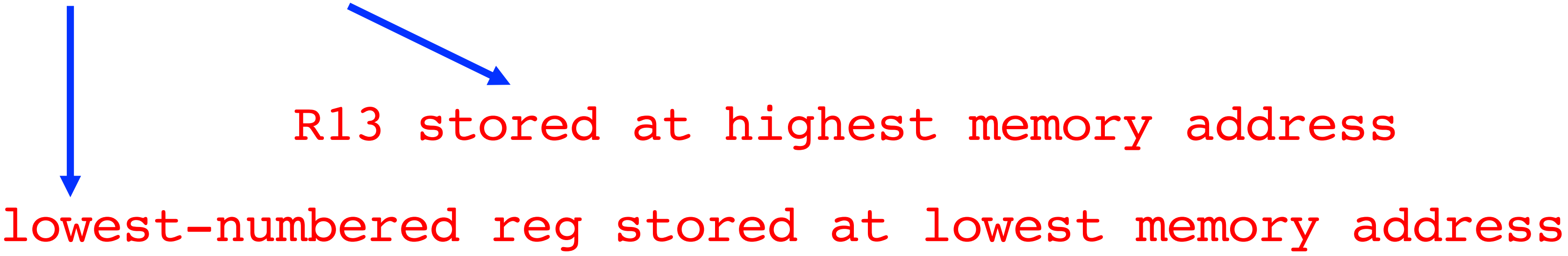
Rules for Caller and Callee

- Caller save rule: The caller must save any non-preserved registers that it needs after the call
  - **After** the call, it must **restore** these registers

- Callee save rule: Before a callee disturbs any of the preserved registers, it must save these registers
  - **Before** the return, it must **restore** these registers

PUSH and POP Instructions

- **PUSH:** Saves registers on the stack
  - `PUSH {R4}` stores `R4` on to the stack and **adds 4 to** `SP`
- **POP:** Restores registers from the stack
  - `POP {R4}` stores `[SP]` in `R4` and **subtracts 4 from** `SP`
- Can store multiple registers on the stack in a single PUSH
  - `PUSH {R4, R8, LR}`



R13 stored at highest memory address

lowest-numbered reg stored at lowest memory address

C Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);

  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}

int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH  {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH  {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP   {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP   {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

ARM Assembly Code

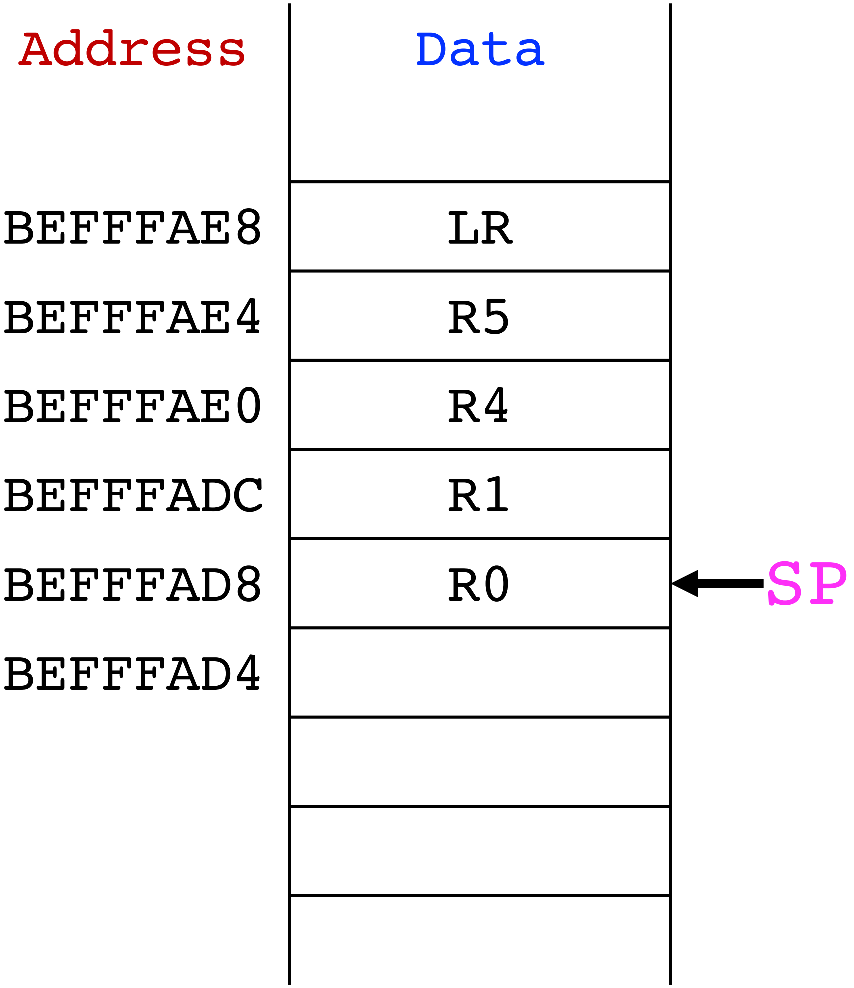
```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR} ; save regs
  ADD   R5,  R0, R1  ; x = (a+b)
  SUB   R12, R0, R1  ; temp = (a-b)
  MUL   R5,  R5, R12 ; x = x*temp
  MOV   R4,  #0      ; i = 0
FOR
  CMP   R4, R0       ; i < a?
  BGE   RETURN       ; no: exit loop
  PUSH {R0, R1}      ; save regs
  ADD   R0, R1, R4   ; arg is b+i
  BL    F2           ; call f2(b+i)
  ADD   R5, R5, R0   ; x = x+f2(b+i)
  POP  {R0, R1}      ; restore regs
  ADD   R4, R4, #1   ; i++
  B     FOR          ; repeat loop
RETURN
  MOV   R0, R5       ; return x
  POP  {R4, R5, LR}  ; restore regs
  MOV   PC, LR       ; return
```

```
; R0=p, R4=r
F2
  PUSH {R4}          ; save regs
  ADD   R4, R0, 5    ; r = p+5
  ADD   R0, R4, R0   ; return r+p
  POP  {R4}          ; restore regs
  MOV   PC, LR       ; return
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH   {R4,  R5, LR}
   ADD    R5,   R0, R1
   SUB    R12, R0, R1
   MUL    R5,   R5, R12
   MOV    R4,   #0
FOR
   CMP    R4, R0
   BGE    RETURN
   PUSH   {R0, R1}
   ADD    R0, R1, R4
   BL     F2
   ADD    R5, R5, R0
   POP    {R0, R1}
   ADD    R4, R4, #1
   B      FOR
RETURN
   MOV    R0, R5
   POP    {R4, R5, LR}
   MOV    PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD    R4, R0, 5
   ADD    R0, R4, R0
   POP   {R4}
   MOV    PC, LR
```



Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	
BEFFFADC	R1	
BEFFFAD8	R0	SP
BEFFFAD4		

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH  {R4,  R5, LR}
   ADD   R5,   R0, R1
   SUB   R12, R0, R1
   MUL   R5,   R5, R12
   MOV   R4,   #0
FOR
   CMP   R4, R0
   BGE   RETURN
   PUSH  {R0, R1}
   ADD   R0, R1, R4
   BL    F2
   ADD   R5, R5, R0
   POP   {R0, R1}
   ADD   R4, R4, #1
   B     FOR
RETURN
   MOV   R0, R5
   POP   {R4, R5, LR}
   MOV   PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD   R4, R0, 5
   ADD   R0, R4, R0
   POP  {R4}
   MOV   PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	
BEFFFADC	R1	
BEFFFAD8	R0	← SP
BEFFFAD4	R4	

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH  {R4,  R5, LR}
   ADD   R5,   R0, R1
   SUB   R12,  R0, R1
   MUL   R5,   R5, R12
   MOV   R4,   #0
FOR
   CMP   R4, R0
   BGE   RETURN
   PUSH  {R0, R1}
   ADD   R0, R1, R4
   BL    F2
   ADD   R5, R5, R0
   POP   {R0, R1}
   ADD   R4, R4, #1
   B     FOR
RETURN
   MOV   R0, R5
   POP   {R4, R5, LR}
   MOV   PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD   R4, R0, 5
   ADD   R0, R4, R0
   POP  {R4}
   MOV   PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	← SP
BEFFFADC	R1	
BEFFFAD8	R0	
BEFFFAD4	R4	

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
   PUSH  {R4,  R5, LR}
   ADD   R5,   R0, R1
   SUB   R12, R0, R1
   MUL   R5,   R5, R12
   MOV   R4,   #0
FOR
   CMP   R4, R0
   BGE   RETURN
   PUSH  {R0, R1}
   ADD   R0, R1, R4
   BL    F2
   ADD   R5, R5, R0
   POP   {R0, R1}
   ADD   R4, R4, #1
   B     FOR
RETURN
   MOV   R0, R5
   POP   {R4, R5, LR}
   MOV   PC, LR
```

```
; R0=p, R4=r
F2
   PUSH {R4}
   ADD   R4, R0, 5
   ADD   R0, R4, R0
   POP  {R4}
   MOV   PC, LR
```

Address	Data	
BEFFFAE8	LR	
BEFFFAE4	R5	
BEFFFAE0	R4	← SP
BEFFFADC	R1	
BEFFFAD8	R0	
BEFFFAD4	R4	

Exercise

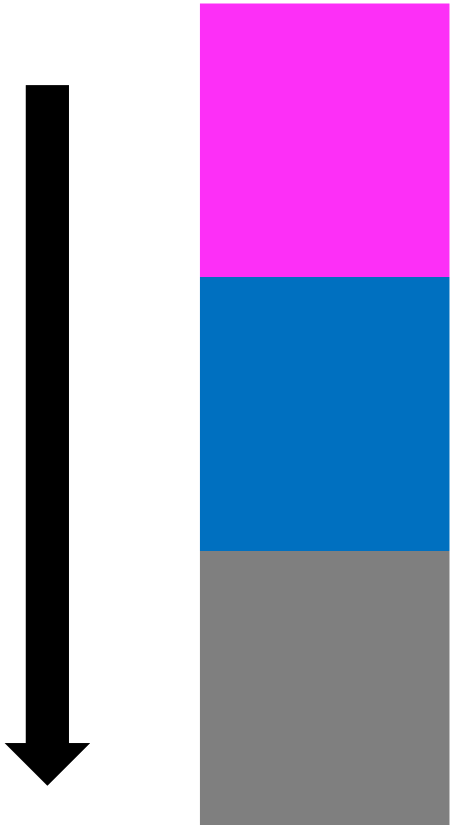
- Provide two optimizations that reduce the stack space consumed by the previous program without impacting its correctness.

Recall: Function Execution

- `f1()`
- `f1()` → `f2()`
- `f1()` → `f2()` → `f3()`
- `f1()` → `f2()`
- `f1()`

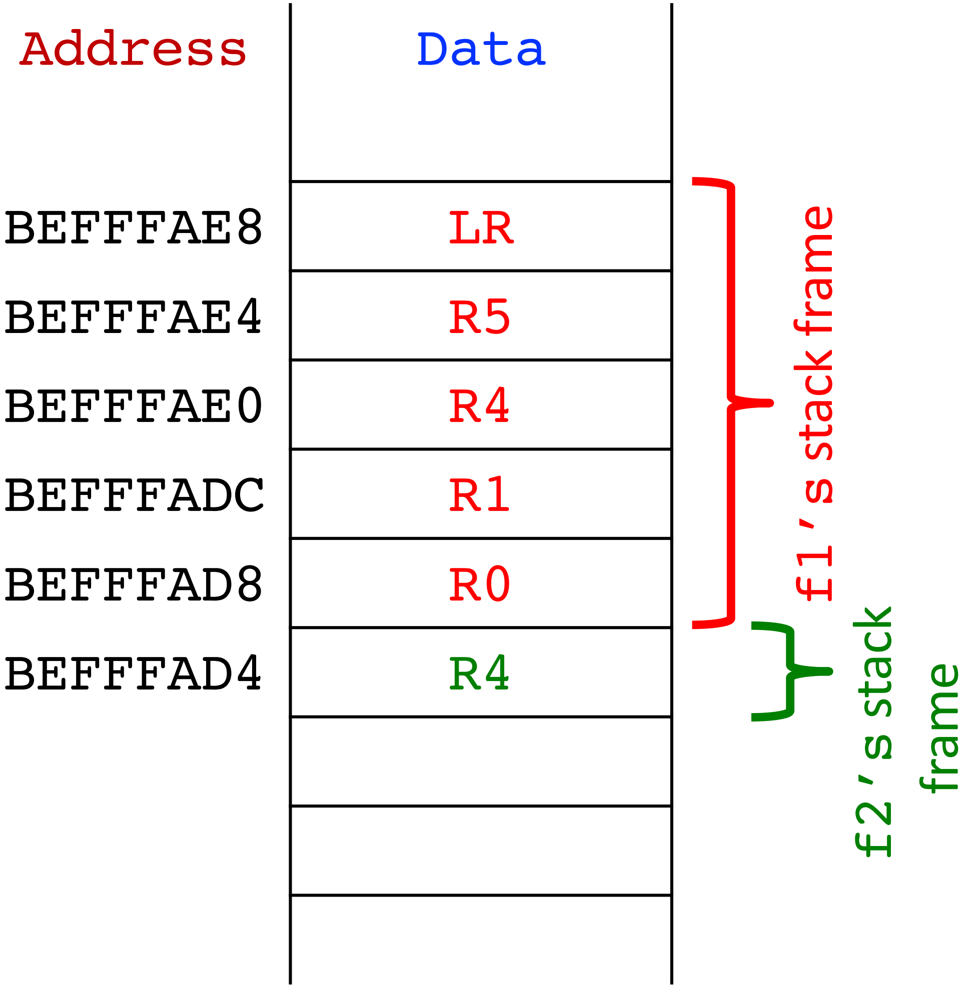
- Stack grows **downward** on function calls

- Stack shrink **upward** as functions **return**



Stack Frame

- The space that a function allocates on the stack is called its stack frame
  - Also called “activation record”
- **Execution Environment of function:** Stack frame, PC, preserved registers
- Caller’s **execution env** must be preserved b/w call & return
- Callee’s **execution env** must be installed on function invocation/activation

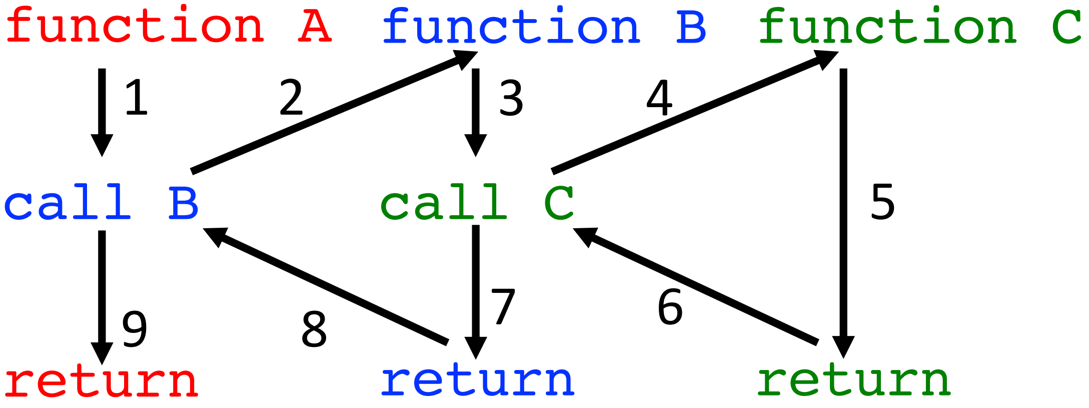


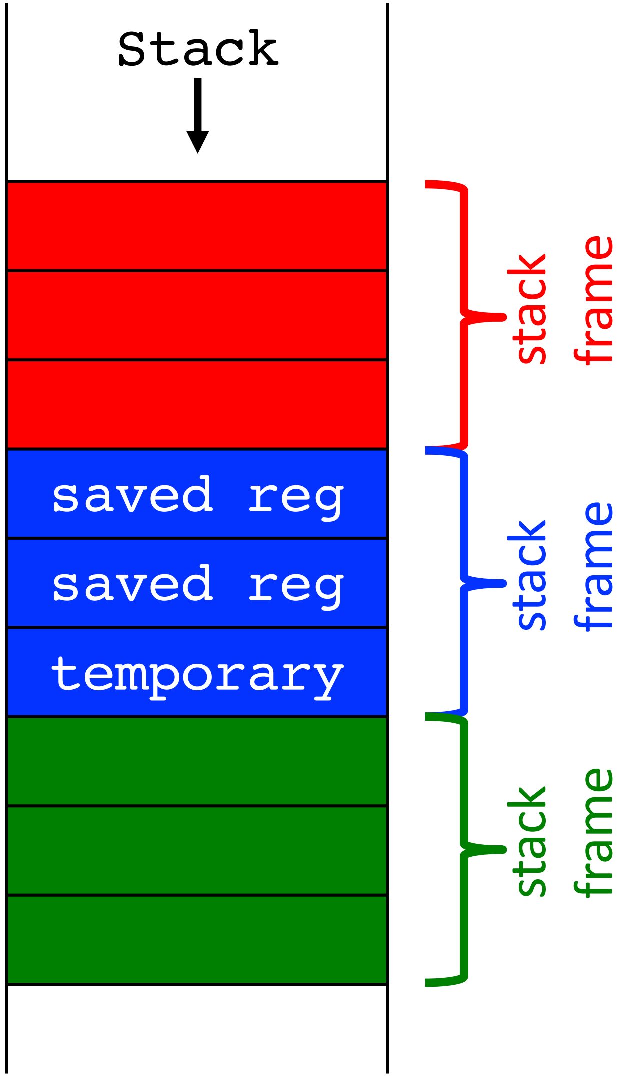
Address	Data
BEFFFAE8	LR
BEFFFAE4	R5
BEFFFAE0	R4
BEFFFADC	R1
BEFFFAD8	R0
BEFFFAD4	R4

Stack Frame

- Many active frames during program execution

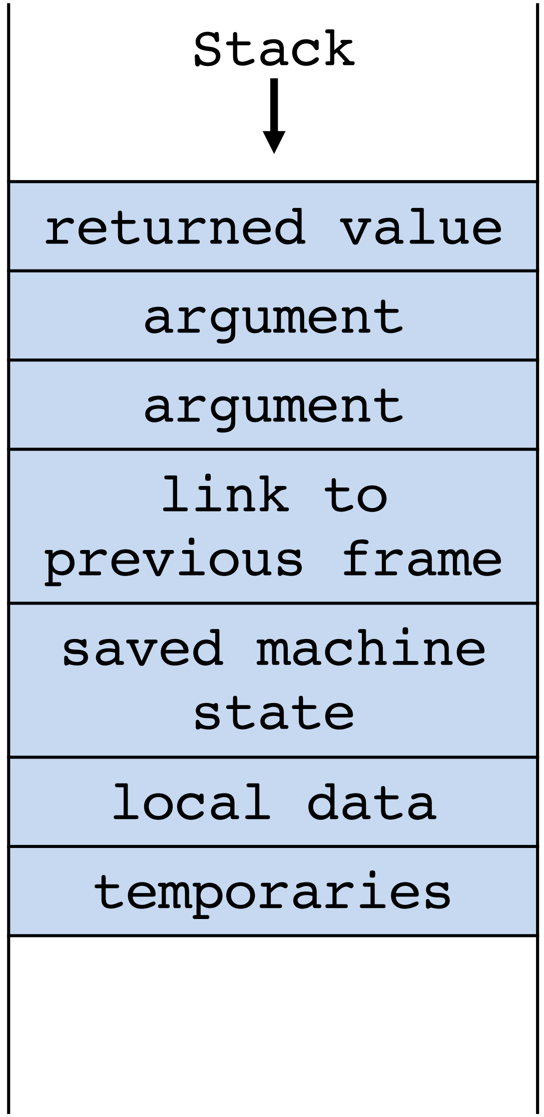
- **We call it the program's call stack**





Things to Remember

- The precise nature & layout of call stack depends on the compiler and architecture
- Stack is **not a hardware component**
- We **set aside** an area in memory and treat it as a stack
    - A different (more generic) stack frame is shown to the right



Application Binary Interface

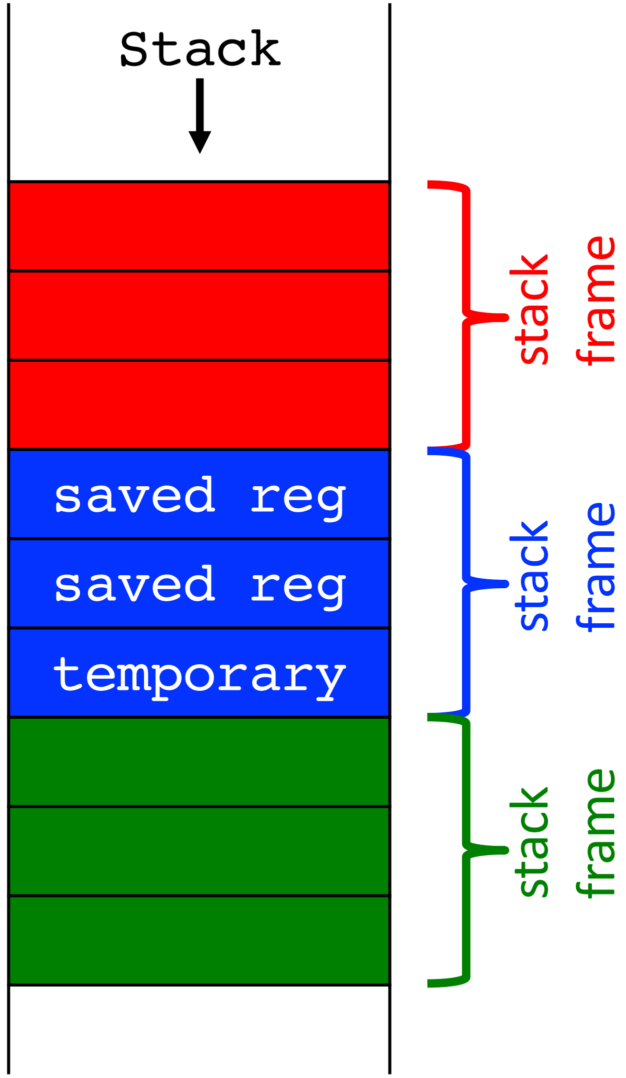
- Calling convention is not part of ISA

- It is part of procedure call interface

- Such aspects make up the **Application Binary Interface** or **ABI**

Group of Stack Frames

- Many names for **the call stack**
  - Execution Stack
  - Program Stack
  - Run-time Stack
  - Control Stack
  - Machine Stack
  - Activation Stack



Summary

- **Caller**
  - Puts arguments in `R0-R3`
  - Saves any needed registers (`R0-R3, R12`)
  - Call function: `BL CALLEE`
  - Restores registers
  - Looks for result in `R0`

- **Callee**
  - Saves registers that might be disturbed `(R4-R11, LR`)
  - Executes the function body (a.k.a. performs the function)
  - Puts the result in `R0`
  - Restores registers
  - Returns: `MOV PC, LR`

Recursion

- Recursion is a **powerful programming technique**
  - Clarity, simplicity, and convenience
- A recursive function is a non-leaf that calls itself
  - Both caller and callee at the same time

```
n = 0, factorial(0) = 1
n = 1, factorial(1) = 1
n = 2, factorial(2) = 2
n = 3, factorial(3) = 6
n = 4, factorial(4) = 24
n = 5, factorial(5) = 120
n = 6, factorial(6) = 720
and so on ....
```

C Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

factorial(3)

C Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

```
n = 3, factorial(3) = 3 * factorial(2)
                    = 3 * 2 * factorial(1)
                    = 3 * 2 * 1 * factorial(0)
                    = 3 * 2 * 1 * 1
                    = 6
```

Recursion

ARM Assembly Code

```
0x8500 FACTORIAL   PUSH  {R0, LR}     ;Push n and LR on stack
0x8504             CMP   R0, #1       ;R0 <= 1?
0x8508             BGT   ELSE         ;no: branch to else
0x850C             MOV   R0, #1       ;otherwise, return 1
0x8510             ADD   SP, SP, #8   ;restore SP
0x8514             MOV   PC, LR       ;return
0x8518 ELSE        SUB   R0, R0, #1   ;n = n - 1
0x851C             BL    FACTORIAL    ;recursive call
0x8520             POP   {R1, LR}     ;pop n (into R1) and LR
0x8524             MUL   R0, R1, R0   ;R0 = n*factorial(n-1)
0x8528             MOV   PC, LR       ;return
```

factorial(3)

ARM Assembly Code

```
0x8500 FACTORIAL     PUSH   {R0, LR}
0x8504               CMP    R0, #1
0x8508               BGT    ELSE
0x850C               MOV    R0, #1
0x8510               ADD    SP, SP, #8
0x8514               MOV    PC, LR
0x8518 ELSE          SUB    R0, R0, #1
0x851C               BL     FACTORIAL
0x8520               POP    {R1, LR}
0x8524               MUL    R0, R1, R0
0x8528               MOV    PC, LR
```

LR 0x1000

R0 0x0003

Address	Data	
BEFFFAE8		← SP
BEFFFAE4		
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

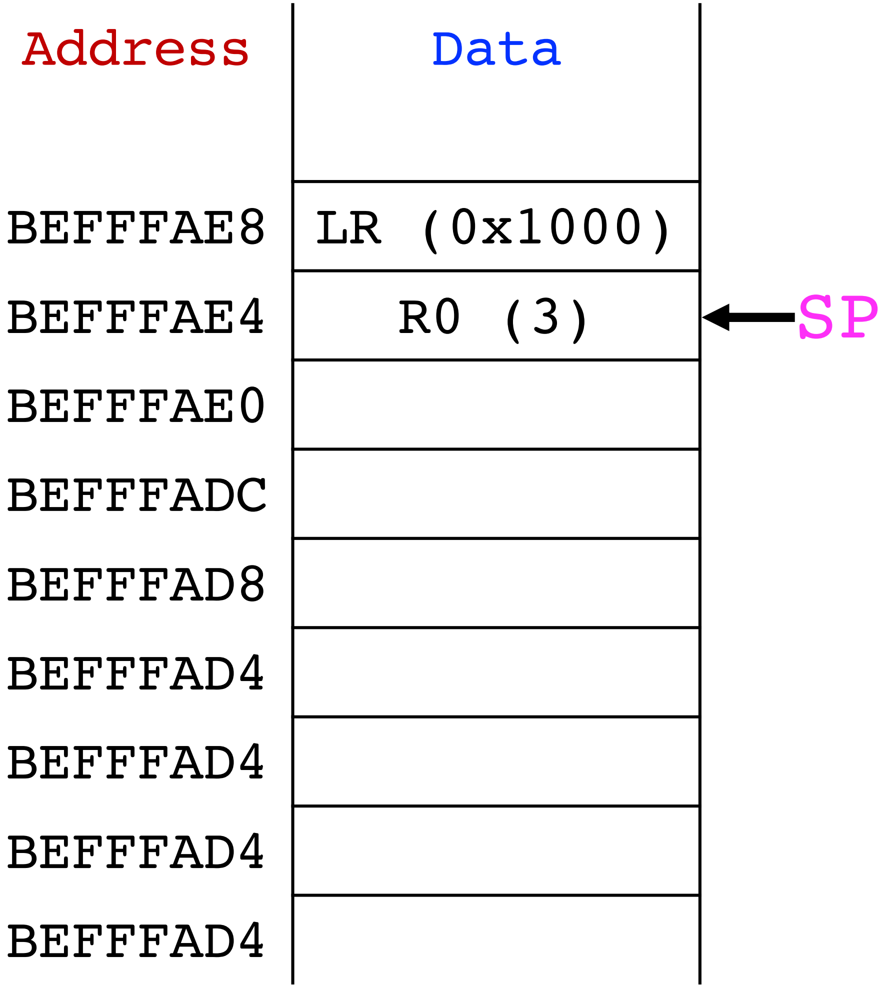
factorial(3)

ARM Assembly Code

```
0x8500 FACTORIAL     PUSH   {R0, LR}
0x8504               CMP    R0, #1
0x8508               BGT    ELSE
0x850C               MOV    R0, #1
0x8510               ADD    SP, SP, #8
0x8514               MOV    PC, LR
0x8518 ELSE          SUB    R0, R0, #1
0x851C               BL     FACTORIAL
0x8520               POP    {R1, LR}
0x8524               MUL    R0, R1, R0
0x8528               MOV    PC, LR
```

LR 0x1000

R0 0x0003



Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	← SP
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(2)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH   {R0, LR}
0x8504              CMP    R0, #1
0x8508              BGT    ELSE
0x850C              MOV    R0, #1
0x8510              ADD    SP, SP, #8
0x8514              MOV    PC, LR
0x8518 ELSE         SUB    R0, R0, #1
0x851C              BL     FACTORIAL
0x8520              POP    {R1, LR}
0x8524              MUL    R0, R1, R0
0x8528              MOV    PC, LR
```

LR 0x8520

R0 0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	← SP
BEFFFAE0		
BEFFFADC		
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(2)

ARM Assembly Code

```
0x8500 FACTORIAL     PUSH   {R0, LR}
0x8504               CMP    R0, #1
0x8508               BGT    ELSE
0x850C               MOV    R0, #1
0x8510               ADD    SP, SP, #8
0x8514               MOV    PC, LR
0x8518 ELSE          SUB    R0, R0, #1
0x851C               BL     FACTORIAL
0x8520               POP    {R1, LR}
0x8524               MUL    R0, R1, R0
0x8528               MOV    PC, LR
```

LR 0x8520

R0 0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH   {R0, LR}
0x8504              CMP    R0, #1
0x8508              BGT    ELSE
0x850C              MOV    R0, #1
0x8510              ADD    SP, SP, #8
0x8514              MOV    PC, LR
0x8518 ELSE         SUB    R0, R0, #1
0x851C              BL     FACTORIAL
0x8520              POP    {R1, LR}
0x8524              MUL    R0, R1, R0
0x8528              MOV    PC, LR
```

LR 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL     PUSH  {R0, LR}
0x8504               CMP   R0, #1
0x8508               BGT   ELSE
0x850C               MOV   R0, #1
0x8510               ADD   SP, SP, #8
0x8514               MOV   PC, LR
0x8518 ELSE          SUB   R0, R0, #1
0x851C               BL    FACTORIAL
0x8520               POP   {R1, LR}
0x8524               MUL   R0, R1, R0
0x8528               MOV   PC, LR
```

LR 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	← SP
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

factorial(1)

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	← SP
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 1

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH   {R0, LR}
0x8504              CMP    R0, #1
0x8508              BGT    ELSE
0x850C              MOV    R0, #1
0x8510              ADD    SP, SP, #8
0x8514              MOV    PC, LR
0x8518 ELSE         SUB    R0, R0, #1
0x851C              BL     FACTORIAL
0x8520              POP    {R1, LR}
0x8524              MUL    R0, R1, R0
0x8528              MOV    PC, LR
```

LR 0x8520

PC 0x8520

R0 0x0001

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	← SP
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 2 x 1

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH  {R0, LR}
0x8504              CMP   R0, #1
0x8508              BGT   ELSE
0x850C              MOV   R0, #1
0x8510              ADD   SP, SP, #8
0x8514              MOV   PC, LR
0x8518 ELSE         SUB   R0, R0, #1
0x851C              BL    FACTORIAL
0x8520              POP   {R1, LR}
0x8524              MUL   R0, R1, R0
0x8528              MOV   PC, LR
```

LR 0x8520   PC 0x8520

R0 0x0002   R1 0x0002

Address	Data	
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	← SP
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

R0 = 3 X 2 = 6

ARM Assembly Code

```
0x8500 FACTORIAL    PUSH   {R0, LR}
0x8504              CMP    R0, #1
0x8508              BGT    ELSE
0x850C              MOV    R0, #1
0x8510              ADD    SP, SP, #8
0x8514              MOV    PC, LR
0x8518 ELSE         SUB    R0, R0, #1
0x851C              BL     FACTORIAL
0x8520              POP    {R1, LR}
0x8524              MUL    R0, R1, R0
0x8528              MOV    PC, LR
```

LR 0x1000   PC 0x1000

R0 0x0006   R1 0x0003

Address	Data	
		← SP
BEFFFAE8	LR (0x1000)	
BEFFFAE4	R0 (3)	
BEFFFAE0	LR (0x8520)	
BEFFFADC	R0 (2)	
BEFFFAD8	LR (0x8520)	
BEFFFAD4	R0 (1)	
BEFFFAD4		
BEFFFAD4		
BEFFFAD4		

Is recursion worth the trouble?

- There is an alternative to solving a problem using recursion
  - Any recursive solution has an equivalent iterative solution (**mathematically sound statement**)
  - **Exercise:** Write `factorial(int n)` with an iterative statement

- Overheads of recursion
  - (CPU) Extra branch instructions due to function calls
  - (Memory) Extra memory is consumed by the stack frames

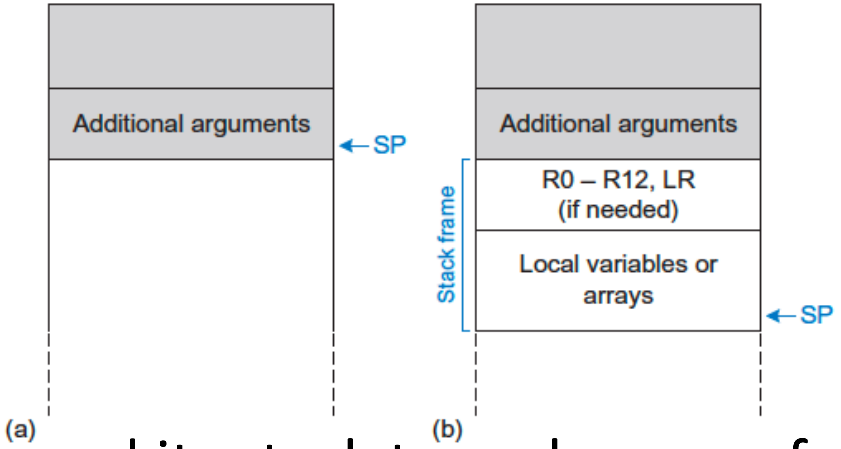
- In many areas, the convenience is worth the trouble
  - Neural networks, data structures, recursive descent parsers

Summary of `factorial`

- `factorial` saves `LR` according to the callee save rule
- `factorial` saves `R0` according to the caller save rule, because it will need **`n`** after calling itself
- if `n is less than or equal to 1` put the result (i.e., 1) in `R0` and return (no need to restore `LR` because it is unchanged)
- Use `R1` for restoring `n`, so as not to overwrite the returned value
- The multiply instruction `(MUL R0, R1, R0)` multiplies `n (in R1)` and the returned value `(in R0)` and puts the result in `R0`

Using Stack for Args & Local Vars

- Functions may have more than four input arguments and may have too many local variables to keep in preserved registers
- The stack is used to store this information



- The caller must expand its stack to make room for additional arguments
- Callee can find the additional arguments in the caller’s stack
  - Exception to the rule that callee must not access caller’s stack

Local Variables and Arrays

- Local variables are declared within a function and can be accessed only within that function (they are stack-resident)

- If there are more local variables than can fit in R4 – R11, they can be stored in the callee's stack frame

- Local arrays are also stored on the stack as they do not fit in registers

Loading Literals

Loading Literals

- Programs need to load 32-bit literals, such as constants or addresses

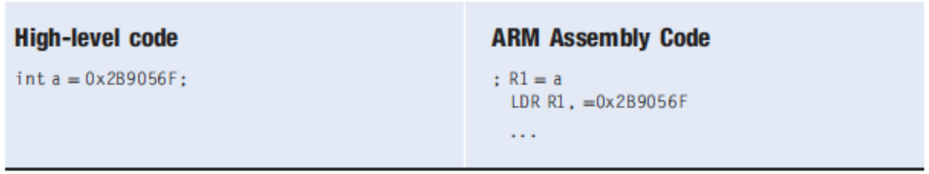
- Each instruction is 32 bits. MOV only accepts a 12-bit constant

- **Solution:** LDR is used to load these numbers from a **literal pool** in the text segment
  - `LDR    Rd,    =literal` → constant
  - `LDR    Rd,    =label` → address

- In both cases, the value to load is kept in a literal pool, which is a portion of text segment containing literals

Loading Literals

- Example ARM assembly



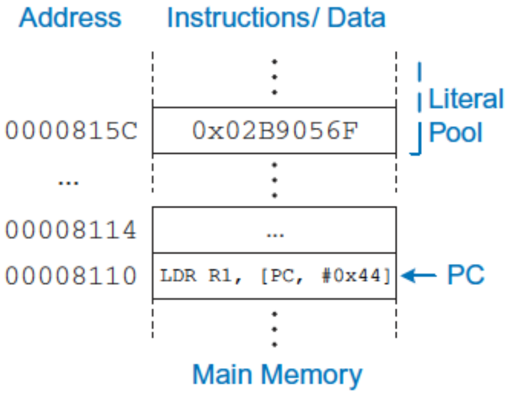
High-level code

```
int a = 0x2B9056F;
```

ARM Assembly Code

```
; R1 = a
  LDR R1, =0x2B9056F
  ...
```

- Caution: The literal pool must be less than 4096 bytes from the LDR instruction so the load can be performed using the base addressing mode



- Systems software (e.g., assemblers and compilers) require **extra care** due to these details

- Must **NOT** accidently point PC to a location inside the literal pool

Call by Value vs. Call by Reference

Argument Passing

- Different high-level languages allow different ways of passing arguments between functions

- How do the different mechanisms **translate** into assembly?

- Two main approaches
  - Call by value
  - Call by reference

- Implications
  - Security and safety
  - Convenience
  - Abstraction

Call by Value

- The actual value is copied into a register or on the stack and is passed to the callee

- The **advantage** of this method is the safety of information hiding
  - Operations performed on the actual arguments do not affect the values in the activation frame of the caller

- The **disadvantage** is the memory required for copies of values requiring large storage

- Default method for passing arguments in C and C++, and the only way in Java

Call by Reference

- Also called called by address or call by location
- A reference of the argument is passed to the function
- Lower overhead for the call and lower memory requirements
- Callee can modify the data belonging to the caller, with possibly serious implications

Call by Value: Registers

```
.data

record:
  .word 100, 200
```

```
; call
LDR   R2, =record
LDR   R0, [R2]
LDR   R1, [R2, #4]
BL    CALLEE

CALLEE:
ADD   R0, #16
; rest of code
```

Call by Value: Stack

```
.data

record:
  .word 100, 200
```

```
; caller’s code
LDR   R2, =record
LDR   R0, [R2]
LDR   R1, [R2, #4]
PUSH  {R0, R1}
BL    CALLEE

CALLEE:
POP   {R0, R1}
ADD   R0, #16
; rest of code
```

Call by Reference

```
.data

record:
  .word 100, 200
```

```
; caller’s code
LDR   R2, =record
BL    CALLEE

CALLEE:
LDR   R0, [R2]
ADD   R0, #16
; rest of code
```

References in HLLs: C and C++

- C and C++ provide special data types for storing and passing references (i.e., addresses of memory locations)

- Having such a data type allows writing low-level code that interfaces directly with devices

- Very high-performance and efficient code

- Unfortunately, a frequent source of memory safety-related errors and security vulnerabilities and bugs

References in HLLs: Java

- There is no concept of reference in Java

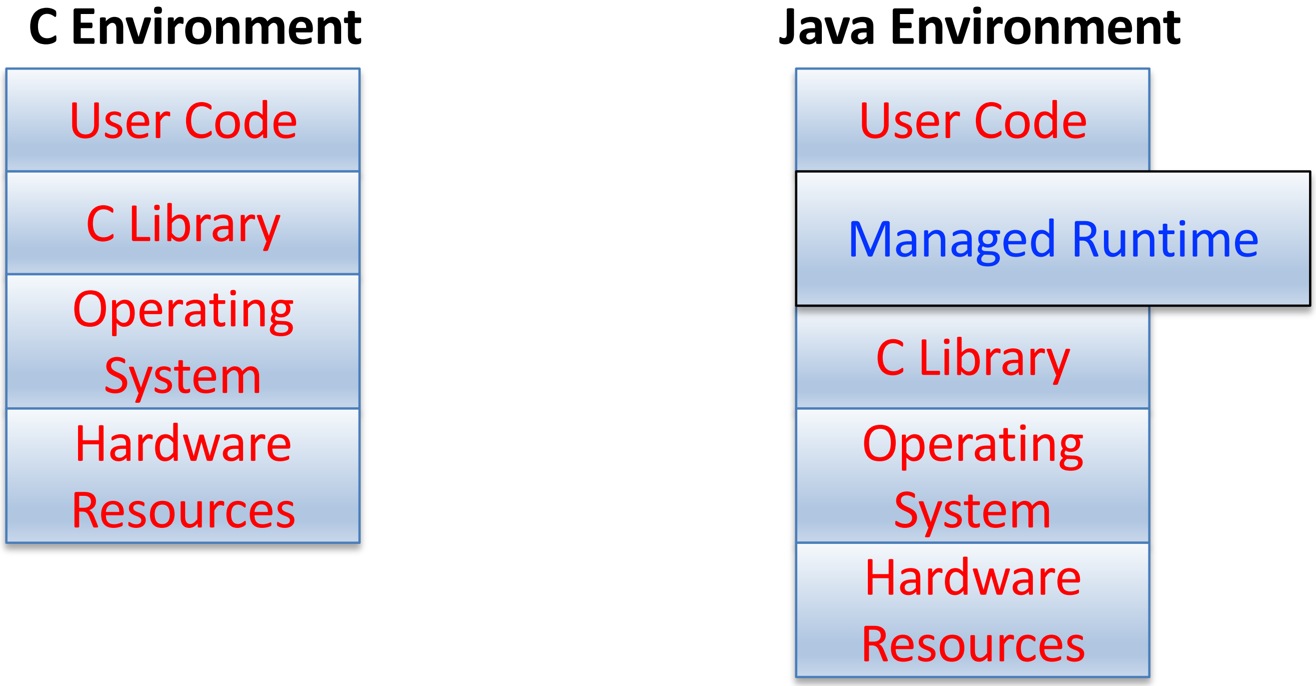
- Only call by value for argument passing

- Application code **NEVER** observes and deals directly with memory addresses

- Address-level manipulations are handled by the runtime environment called the **Java Virtual Machine (JVM)**

Software Stack: C/C++ and Java

- Java, Python, Scala, Ruby are called **managed languages** because memory is managed on behalf of the programmer

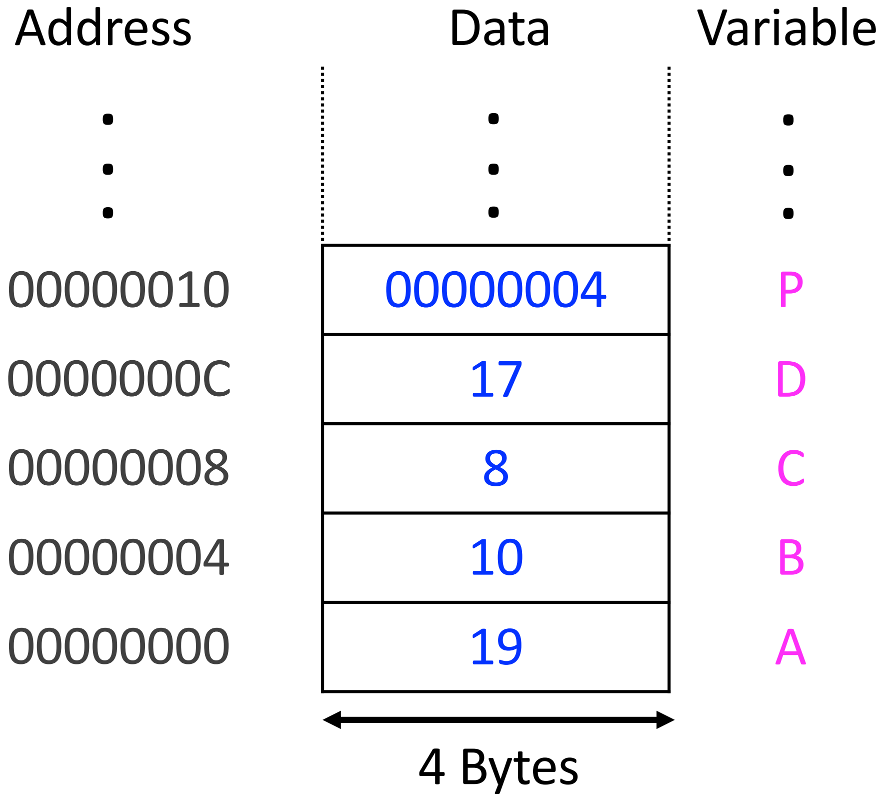


- Each managed language has its own managed runtime
  - Java's runtime environment is called the **J**ava **V**irtual **M**achine (JVM)

Pointers in C (and C++)

- A pointer is a variable that contains the address of another variable (references a location in memory)

```
int A = 19;
int B = 10;
int C = 8;
int D = 17;
....
int *P = &B;
// unary operator & gives
   the address of a
   variable
// P is a reference to B
```

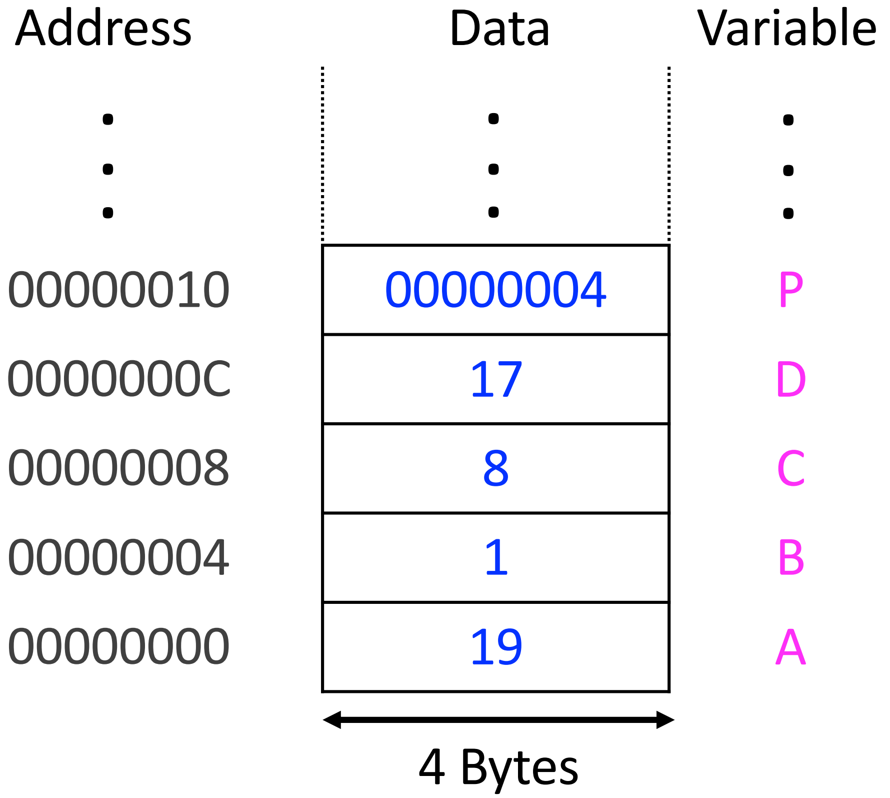


Address	Data	Variable
⋮	⋮	⋮
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	10	B
00000000	19	A

Pointers in C (and C++)

- Can use the pointer to access the value stored in a memory location

```
int A = 19;
int B = 10;
int C = 8;
int D = 17;
....
int *P = &B;
*P = 1;
// * is a dereferencing
   or indirection
   operator that accesses
   the value stored at
   address in P
```

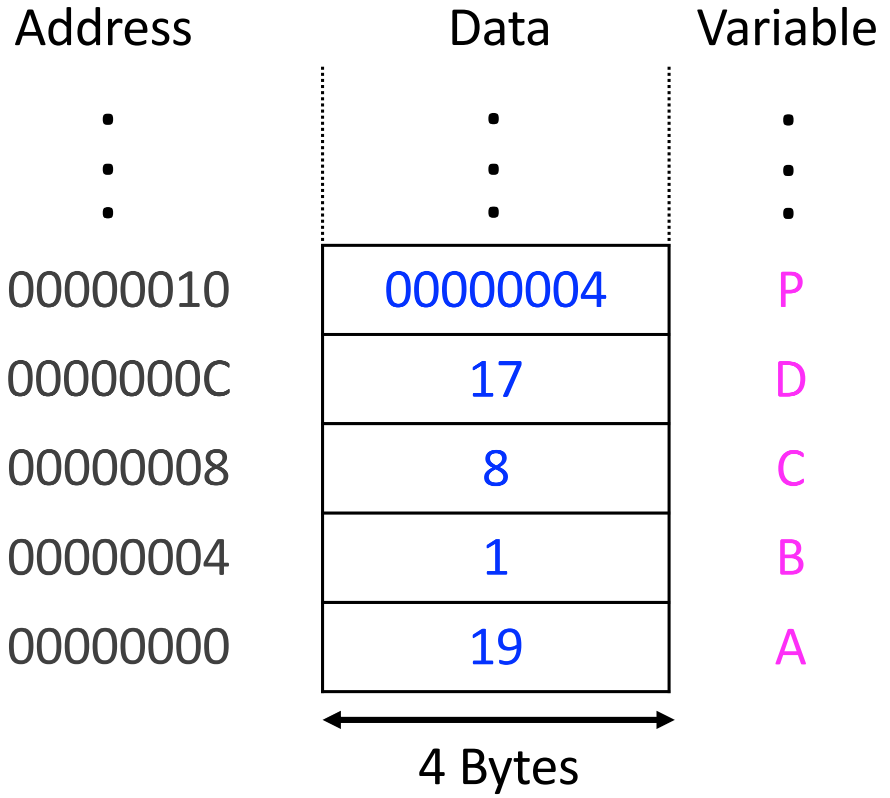


Address	Data	Variable
⋮	⋮	⋮
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	1	B
00000000	19	A

Pointers: Example

- A pointer is 4 bytes on a 32-bit system and 8 bytes on a 64-bit system & it can be stored on the stack or data segment like ordinary variables

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
   00000004
int  x = *P; // x=?
char y = *Q; // y=?
```



Address	Data	Variable
⋮	⋮	⋮
00000010	00000004	P
0000000C	17	D
00000008	8	C
00000004	1	B
00000000	19	A

Pointers

- A pointer points to a memory location and its content is a memory address

- It wears **"datatype glasses"**
  - Wherever it points to, it sees through these glasses

- The variable stored at some memory address can be interpreted via the dereferencing operator (*) as a character or integer or float depending on the type of the pointer

Pointers: The Good

- **Memory efficient** code

- Low-level software code requires access to memory locations
  - Devices are exposed as memory addresses
  - Memory-mapped I/O (more later)

- Programmer has "unlimited power" in managing memory as a resource

Pointers: The Ugly

- Programming is prone to errors

- Bug in manipulating a value – Application produces incorrect output (may be tolerable some of the time)

- C allows pointer arithmetic
  - Bug in updating a memory address – Serious security violation (access violation, corrupted state, and so on)

- C requires programmers to free memory – Memory leaks

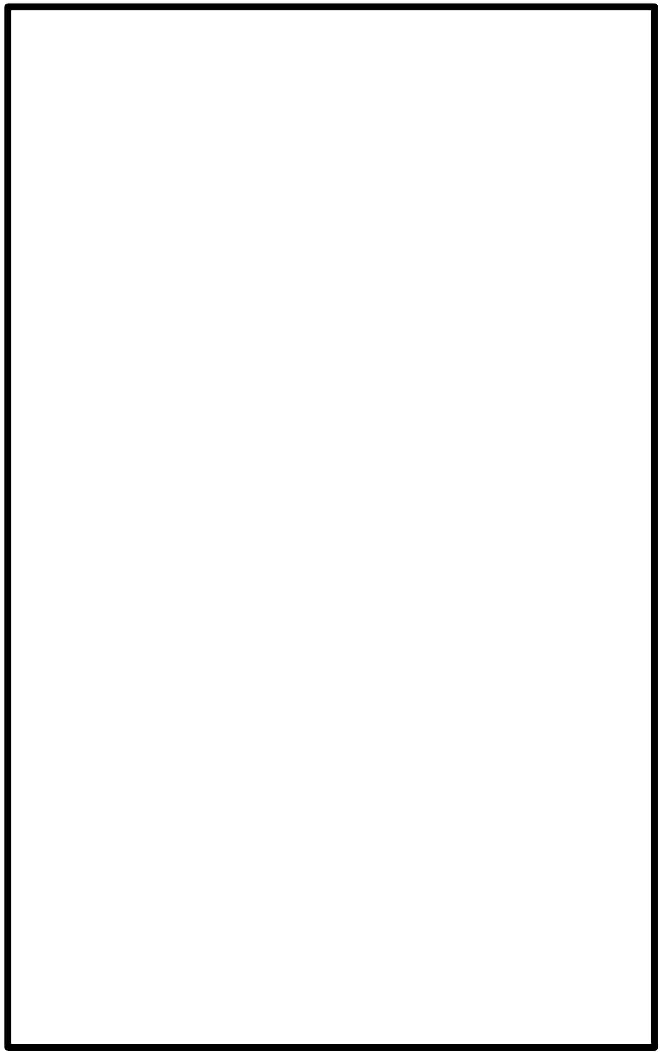
Memory Map

Address Space

- **Address range**
  - A 32-bit (ARM) CPU generates addresses in the range 0 to 0xFFFFFFFC (4294967292)
  - With a 4×10^9 address range, the CPU can access 4 billion individual bytes
- **Address space**
  - The address space of a 32-bit CPU is 2^{32} bytes which equals 4 Gigabytes (GB)

Address Space

- Each word is `32 bits` or `4 bytes`. Address of first & last word is shown
- The address space is empty as shown here
    - Let’s populate with stack and code and data



oxFFFFFFFC

ox00000000

Questions

- Where is the code, data, and the stack in the address space?

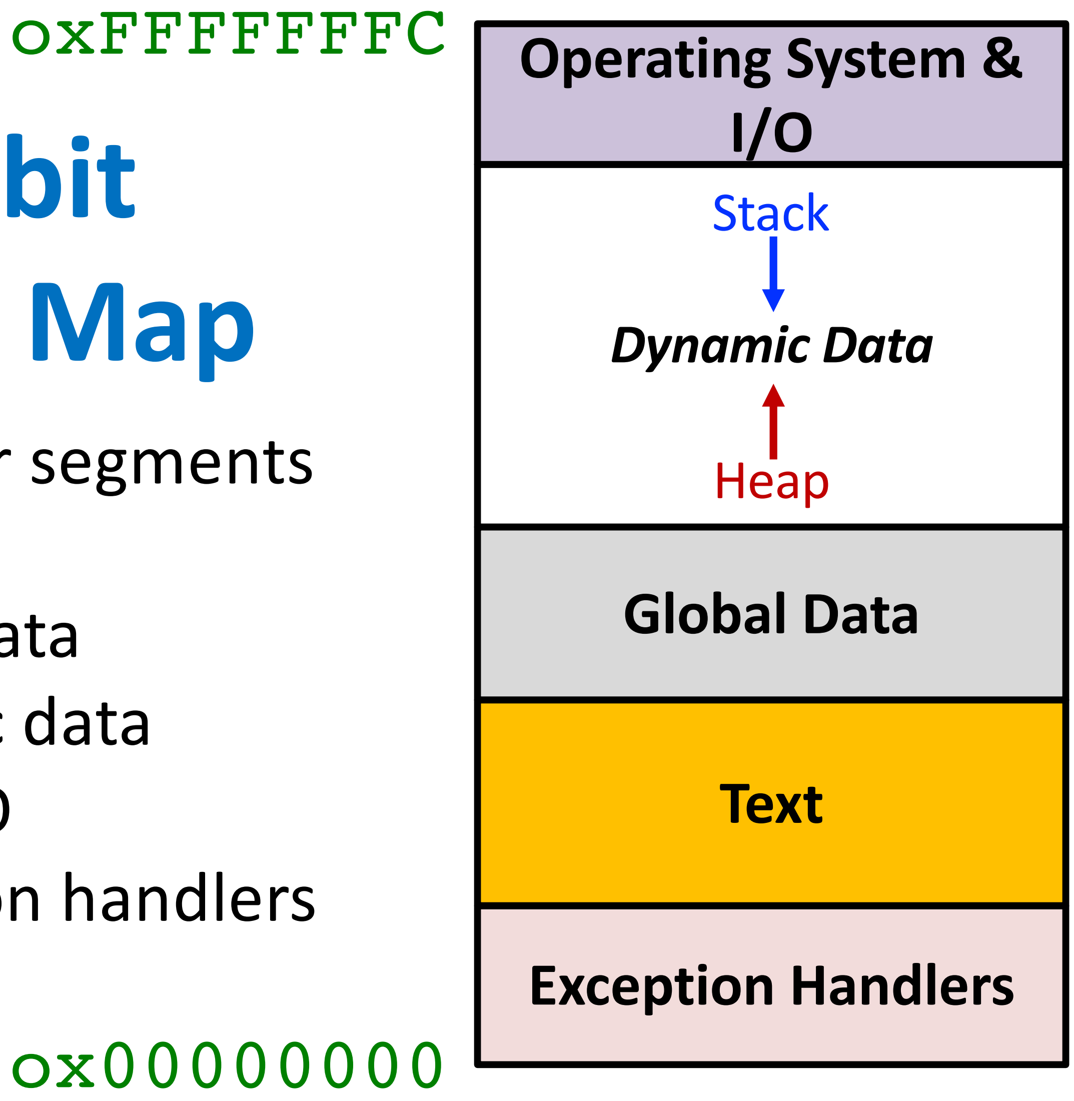
- **Memory map**
  - Defines where code, data, and stack memory are in the program address space

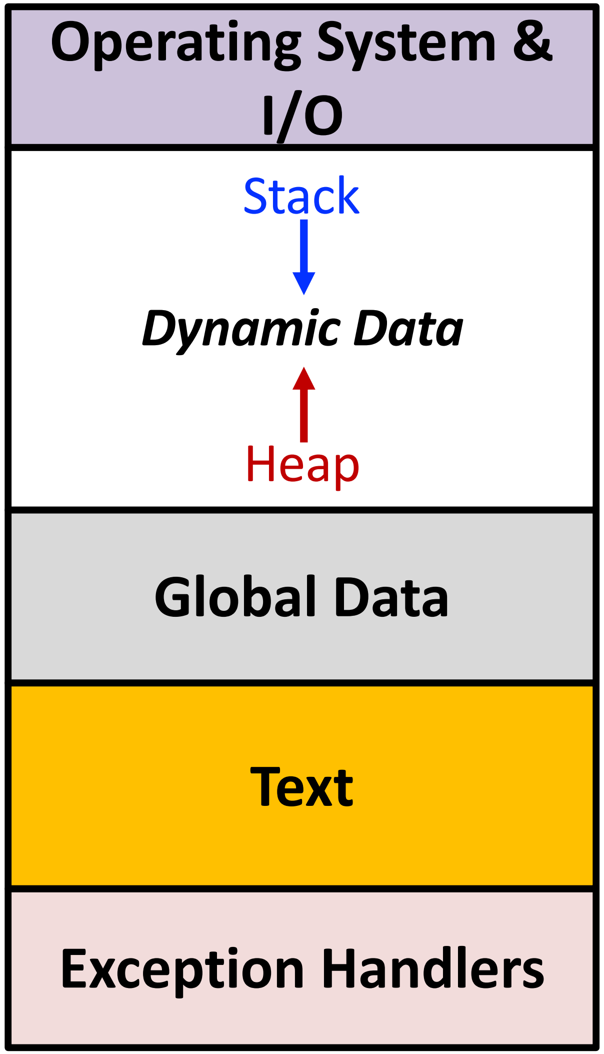
  - Differs from architecture to architecture

  - The subsequent discussion pertains to ARM

ARM 32-bit Memory Map

- Five parts or segments
  - text
  - global data
  - dynamic data
  - OS & I/O
  - Exception handlers





- *Data in this segment is dynamically allocated and deallocated during program execution*
- *Heap data is allocated by the program at run-time*
  - `malloc() and new`
- *Heap grows upward, stack grows downward*

- *Global variables visible to all functions (contrasted with local variables that are only visible to a function)*

- *Machine language program*
- *Also called read-only (**RO**) segment*
- *Literals (constants) such as “Hello”*

ARM Memory Map (with addr ranges)

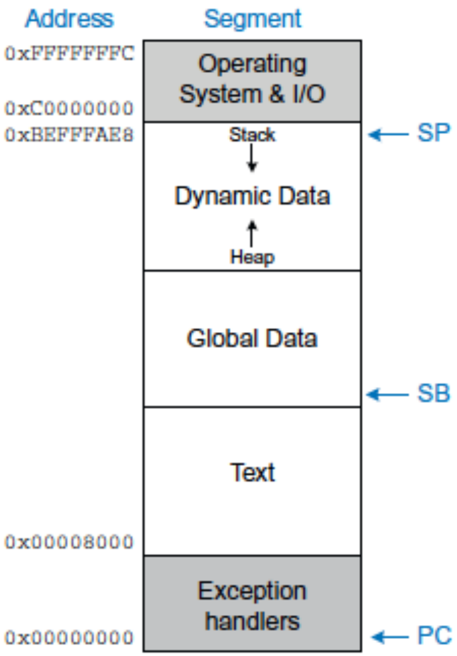


Figure 6.30 **Example ARM memory map**

Virtual Memory

- What if the system has less than 4 GB of physical memory?
- Desktop and high-performance computers use virtual memory
- The address space is virtual (virtual addresses) and it is managed by the operating system
- OS performs virtual to physical address translation

Virtual Memory

- Divide 4 GB of address space into 4 KB pages

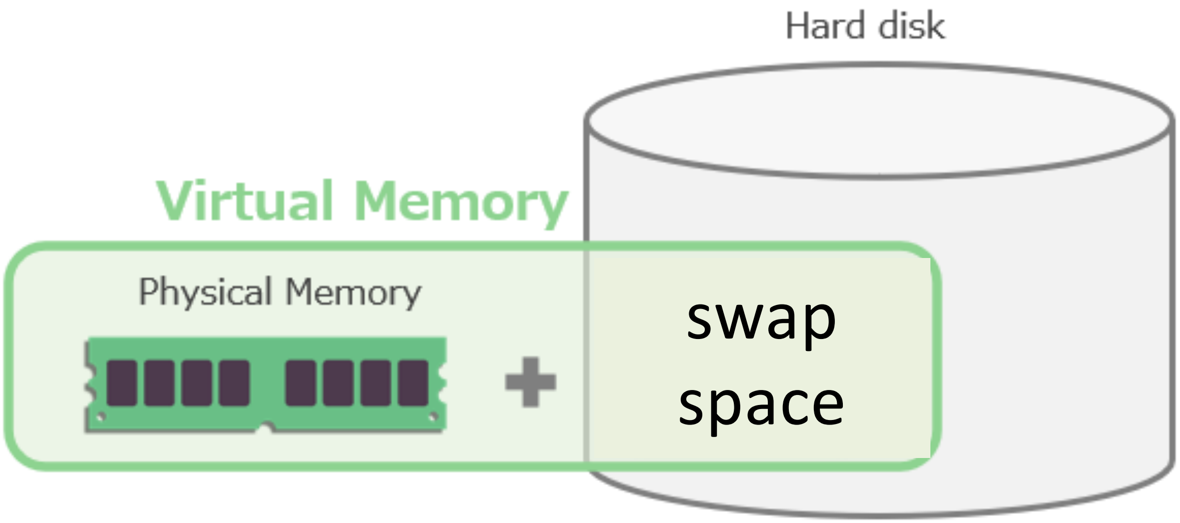
- Each page is a virtual page and has a virtual page number
  - 1, 2, 3, 4, ……

- Each virtual page also has a physical page #
  - Not one to one correspondence

- Operating system allocates physical pages to virtual pages

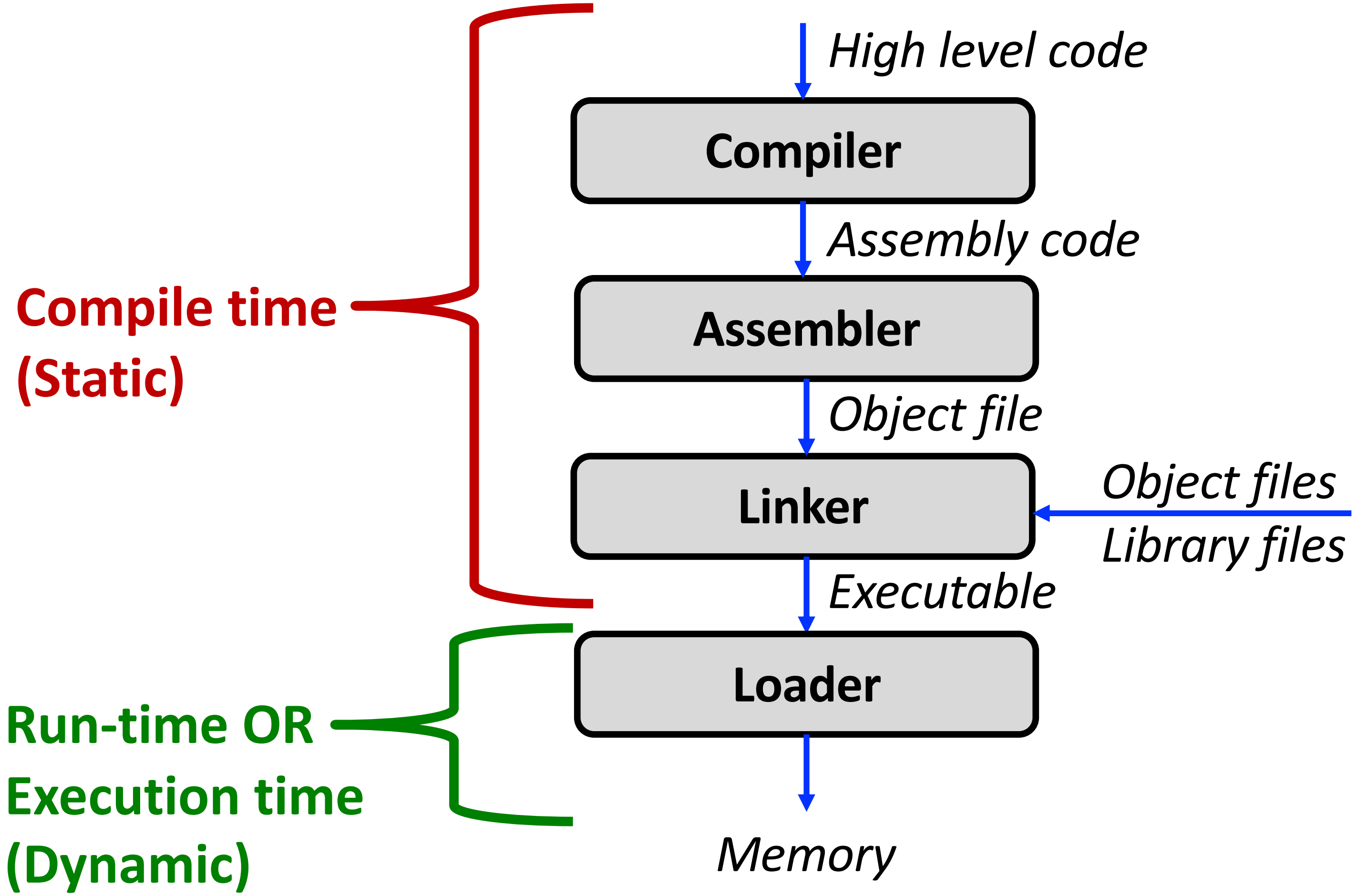
Virtual Memory

- The OS moves a virtual page from memory to a **“swap space”** on disk when it runs out of physical memory
- It **“remembers”** the mapping of virtual page # to either main memory or disk locations in a page table
- A page table maps virtual pages to their location in key-value pairs where key is the virtual page # and value is the location of a page



Starting a Program

Translating and Starting Programs



Heap

Static vs. Dynamic

- Static in computer science means
  - When the program is being compiled
  - At compile time

- Dynamic means
  - During program execution
  - At run-time

- Some aspects of the program behavior are known statically and other only at run-time

Statically Allocated Memory

- Statically allocated memory
    - Global variables
    - Local function variables on the stack
- Compiler knows the following about such variables
    - Their size (which is fixed at compile time)
    - Their location on the stack or global area
    - Their lifetime – either entire program or function execution

Heap

- Heap is for data that is allocated dynamically

- Heap is a large subdividable block of memory

- In C, programmers explicitly allocate and deallocate heap memory

- A piece of code called a “dynamic memory allocator” is used to find free space on the heap memory

Stack vs. Heap

- Space on the stack is allocated and deallocated **“automatically”** in a strict LIFO order

- Sometimes it is necessary to use a variable or array beyond the execution of a function (and keep it alive/around up to some point)

- Sometimes the size of the space to be allocated is not known at compile time or needs to be **resized**

- For all of the above heap is a more convenient space for storage allocation

Deallocation of Memory

- Also called **“freeing memory”**

- DRAM is a limited resource

- Programs run for a **LONG** time in the CPU world (1 sec = 2 billion cycles)

- Allocated memory becomes garbage when it is no longer needed (**but stack space is reclaimed when function returns**)

- It must be recycled for reuse by someone or something else

Heap

- Lifetime of heap variables and arrays extend from allocation until deallocation

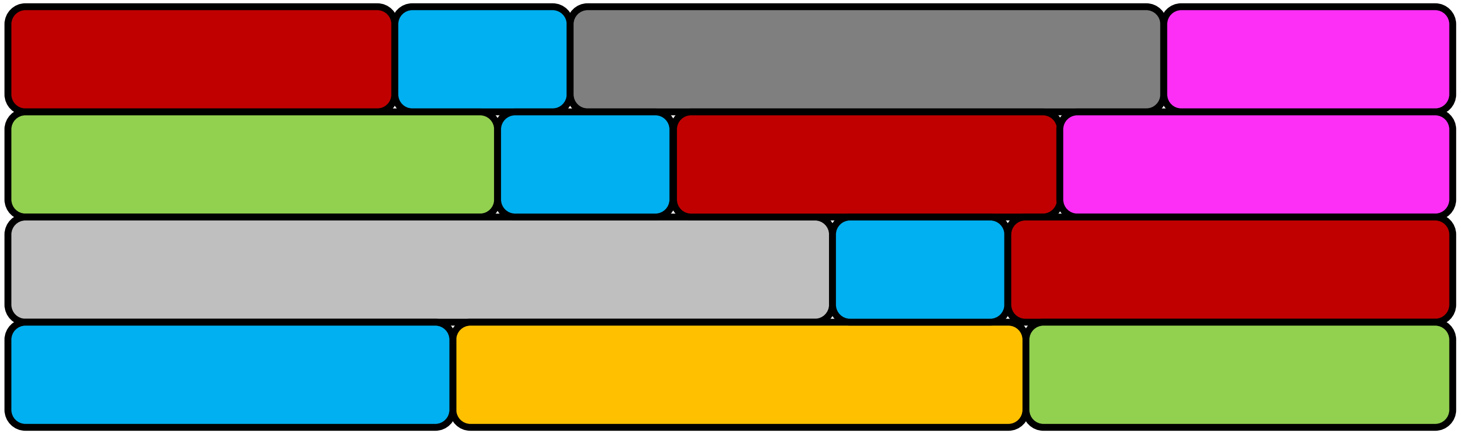
- Memory managers (that include a memory allocator) are libraries that help the programmer in managing the heap

- C memory manager facilitates allocation and deallocation

- **Java and Python:** Allocation same as C, but take "facilitate" to the extreme when it comes to deallocation
  - Does it automatically – **Garbage Collector**!

Heap Organization

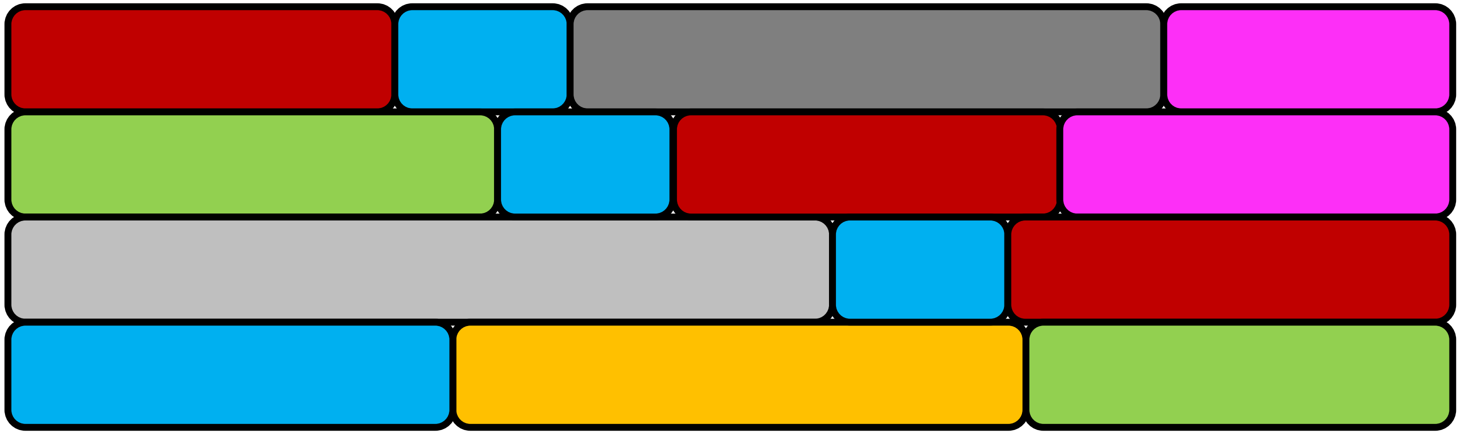
- Programs dynamically allocate objects (arrays, structures) of different types and sizes on the heap over time



- **Heap is now full!**

Heap Organization

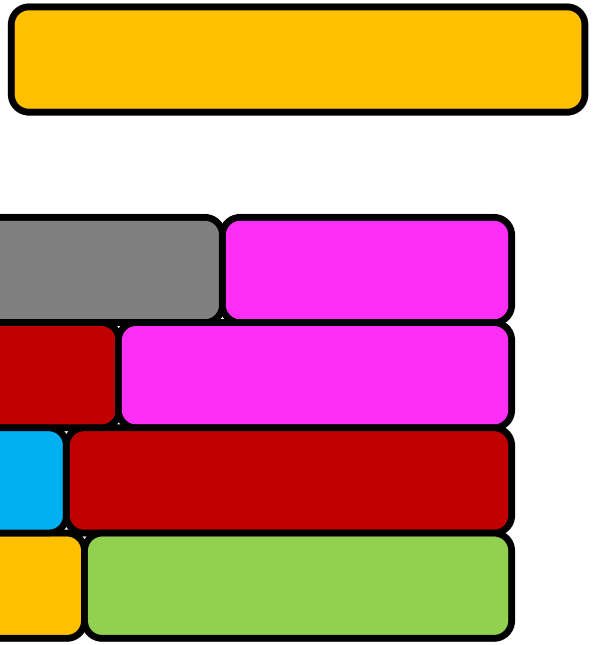
- C programmers need to track lifetime of heap variables
- Suppose we do not need anymore 🟨



- **Heap is now full!**

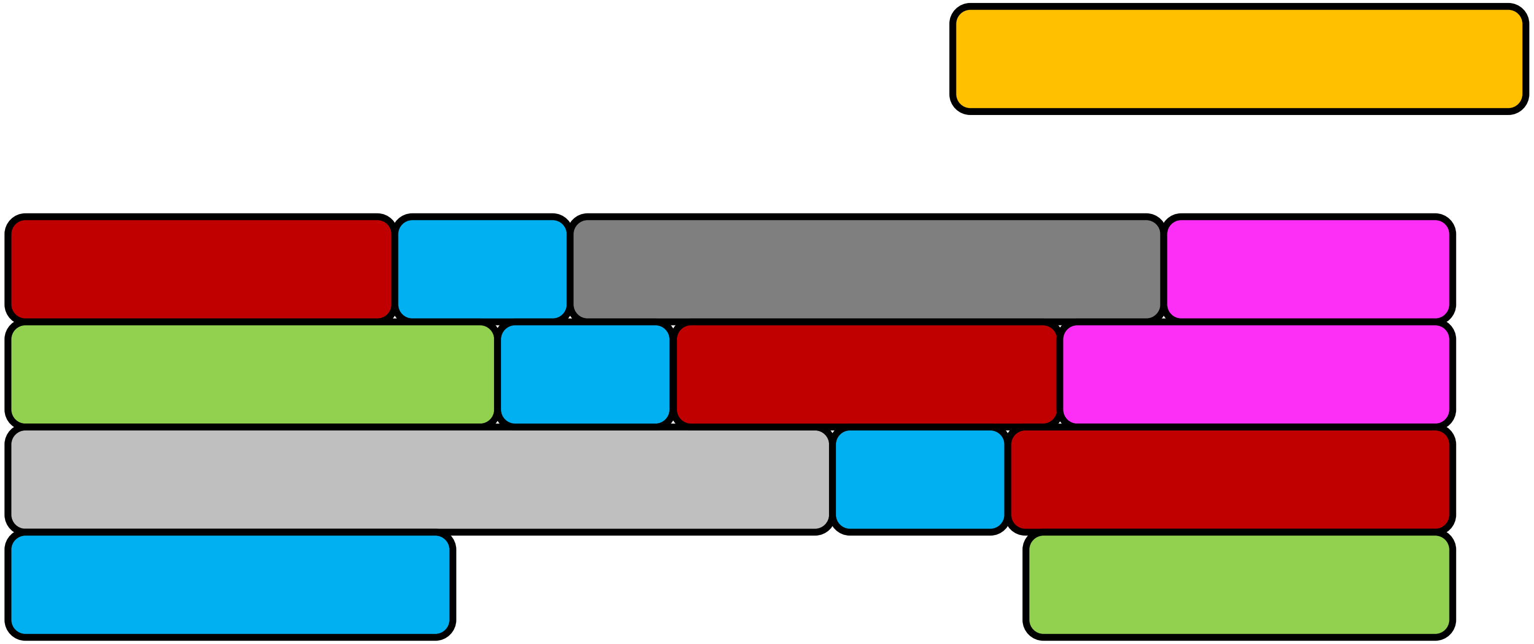
Heap Organization

- Let’s free some space on the heap



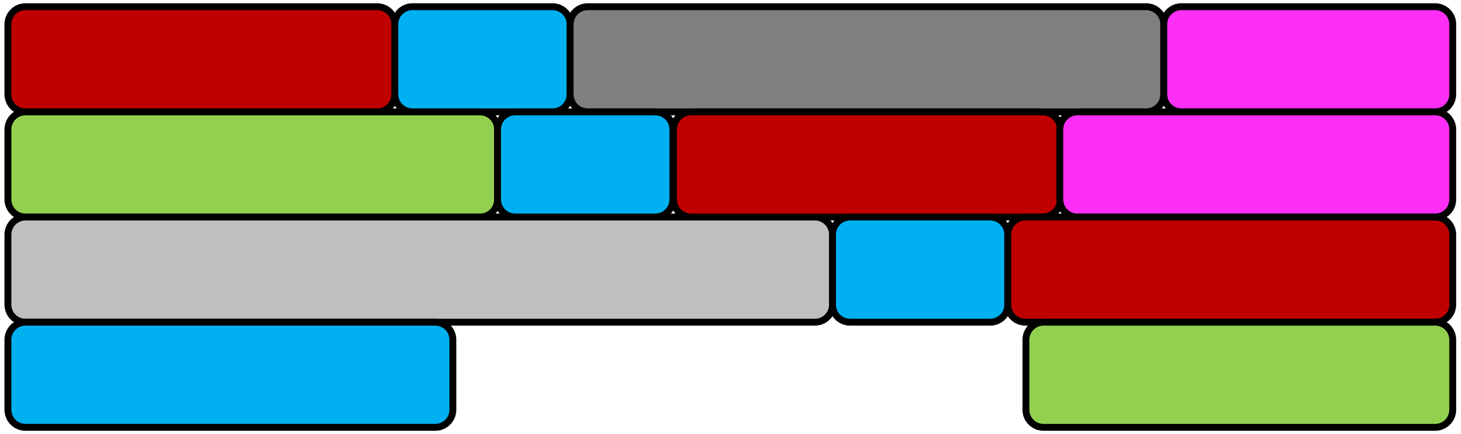
Heap Organization

- Let’s free some space on the heap



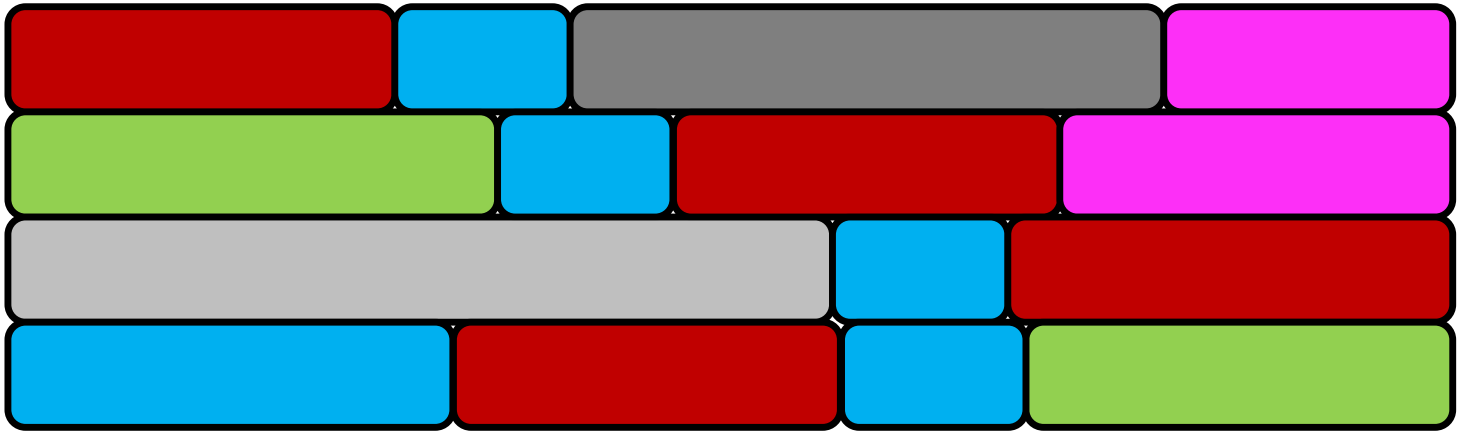
Heap Organization

- Now we can allocate new objects



Heap Organization

- Now we can allocate new objects



Heap Management

- We can deallocate (free) objects in any order (unlike the stack)
- C/C++ programmers need to deallocate objects that are no longer needed
  - Otherwise, heap will remain full even if we can have some free space
- Not returning unused heap back to the memory manager is called a *memory leak*

C Memory Manager: malloc & free

- C library provides malloc (short for memory allocate) and free to allocate and deallocate heap memory, respectively

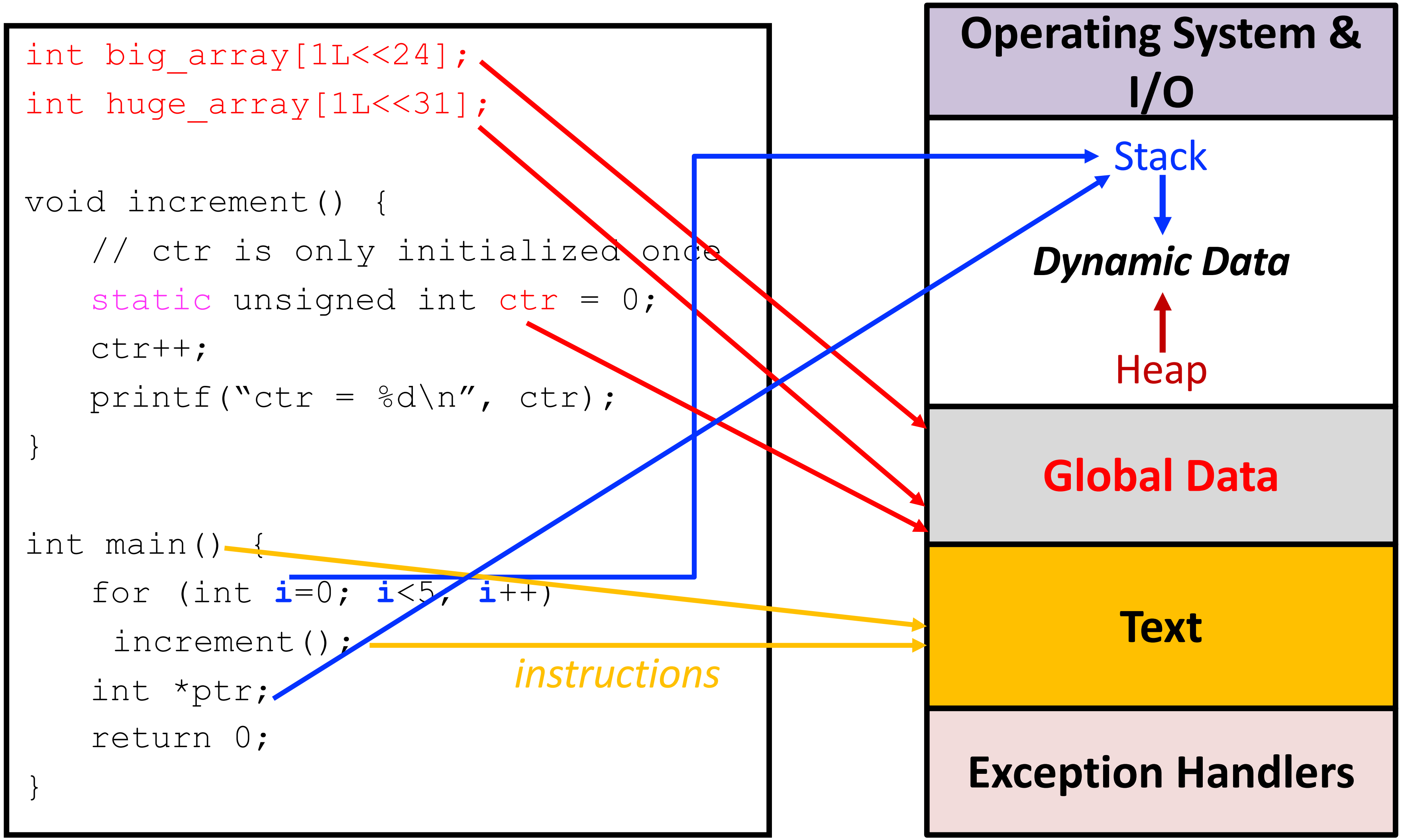
```
#include <stdlib.h>
#include <stdio.h>

void useless_func() {
    int *array = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array[i] = i * i;
    int sum = array[0] + array[9];
    free(array);
    printf(“%i\n”,sum);
    return;
}
```

malloc and free

- malloc
  - Declaration in C library: `void *malloc(size_t size)`
  - Takes input as size (# bytes)
  - Returns a void pointer that can be casted to any pointer type
- free
  - Declaration in C library: `void free(void *ptr)`
  - Memory manager knows how many bytes to free, all it needs is the starting address

Where is everything mapped?



```
int big_array[1L<<24];
int huge_array[1L<<31];

void increment() {
    // ctr is only initialized once
    static unsigned int ctr = 0;
    ctr++;
    printf("ctr = %d\n", ctr);
}

int main() {
    for (int i=0; i<5; i++)
     increment();
    int *ptr;
    return 0;
}
```

Operating System & I/O

Stack

Dynamic Data

Heap

Global Data

Text

instructions

Exception Handlers

A LOT More on this in COMP2310

- Students implement a memory manager from scratch
- Like the CPU: piece by piece
- You write code in C:
  - **Manual memory management for twelve weeks!**
- Enroll and discover all the simplifications (a.k.a. LIES) we have told you this semester

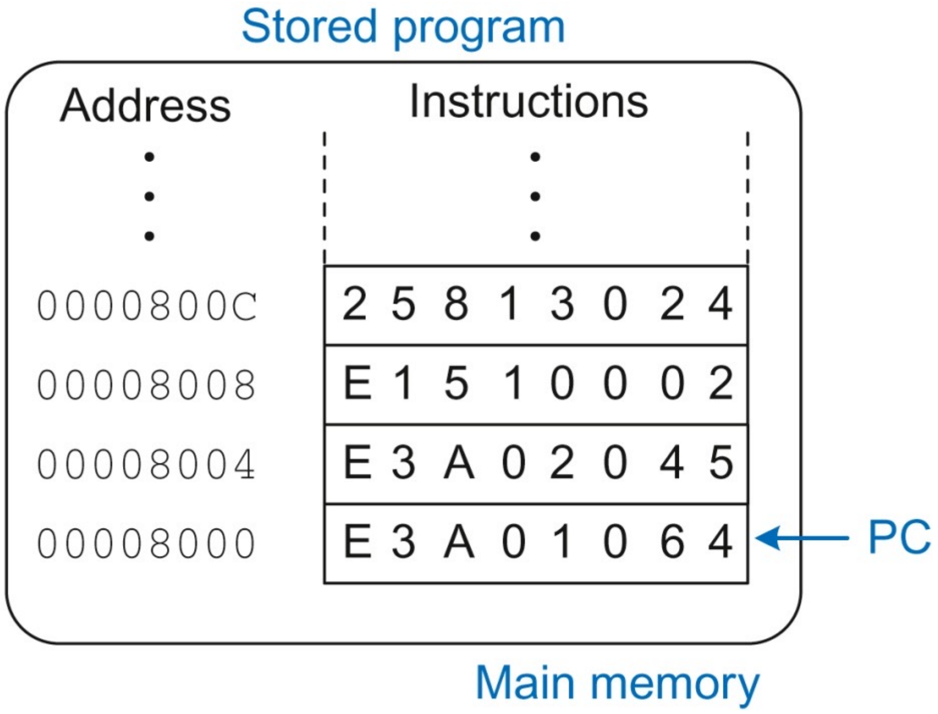
Exceptions

Reading: Section **6.6.3** of H&H

Recall the Stored Program Concept

- When do control flow change from sequential execution to somewhere else?

Assembly code	Machine code
`MOV   R1, #100`	0xE3A01064
`MOV   R2, #69`	0xE3A02045
`CMP   R1, R2`	0xE1510002
`STRHS R3, [R1, #0x24]`	0x25813024



Stored program

Address	Instructions
⋮	⋮
0000800C	2 5 8 1 3 0 2 4
00008008	E 1 5 1 0 0 0 2
00008004	E 3 A 0 2 0 4 5
00008000	E 3 A 0 1 0 6 4

Recall the Stored Program Concept

- When does control flow change from sequential execution to somewhere else?
  - **Unconditional** and **conditional** branches
  - Function **calls** and **returns (also branches)**

- The above change in control flow is due to *changes in program state*

- A useful system must change control flow due to *changes in system state* *(recall computer system = hardware + software)*

Reading: Section **6.6.3** of H&H

Exceptions

- The change in state is known as an event

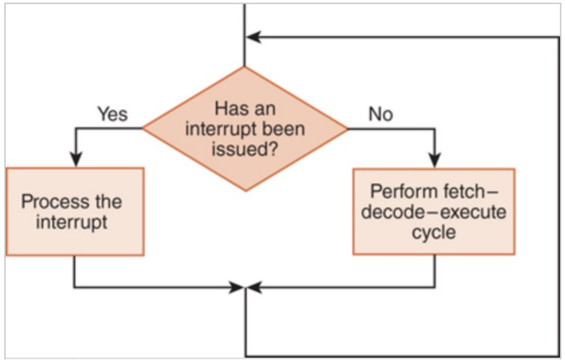
- These changes are “unplanned” events
  - Divide by zero (**execution cannot continue!**)
  - Data arrives from an I/O device (keyboard or disk)
  - An I/O device needs attention
  - System timer expires

- An exception is like an **unscheduled function call** that branches to a new address

- Exceptions may be caused by hardware or software

Interrupts

- Hardware exception **triggered** by an I/O device such as a keyboard is called an interrupt


Has an interrupt been issued?
Yes
No
Process the interrupt
Perform fetch–decode–execute cycle

- CPU receives notification when a user presses a key on the keyboard

- Stop the normal fetch cycle and handle the interrupt

- Like any other function call, the exception must save the return address, jump to some address, do its work, clean up, and return to the program

Example of Interrupts

- I/O device needing service
  - keyboard input, video input

- Periodic system timer expiration

- Power failure

- Machine check: hardware error
  - Bit flip (unplanned!)

Traps

- Software exceptions are also called traps

- Undefined opcode
- Divide by zero or overflow
- Reading from a “bad” memory address (access protection error)
- system call

- **System call** is a form of trap that the program uses to invoke a function in the operating system (OS) running at a higher privilege level

System Call

- An important form of trap which the program uses to invoke a function is the operating system (OS)

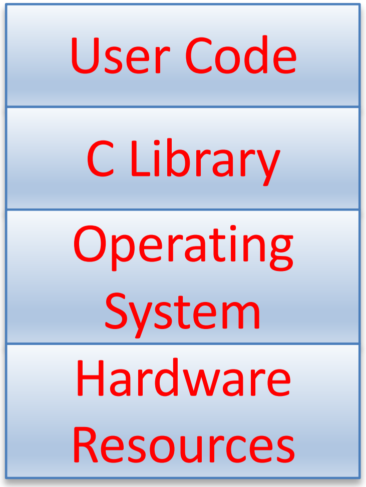
- The function runs at a higher “privilege level”

- Running at a higher privilege level allows access to all system resources

- User/application code (as apposed to OS code) runs at a lower privilege level

Reason for Privilege Levels

- The distinction between privilege levels prevent
  - buggy user code from corrupting other programs
  - crashing the system
  - malicious code from taking over the system
- OS has full control of the system
- Ordinary user code does not!



User Code

C Library

Operating System

Hardware Resources

How should the CPU handle exceptions?

- Both exceptions and interrupts require
  - stopping the current program
  - saving the architectural state
  - handling the exception/interrupt → switch to handler
  - (if possible and make sense) returning back to program execution
- The program branches to a code in the OS that handles the exception

When to handle exceptions?

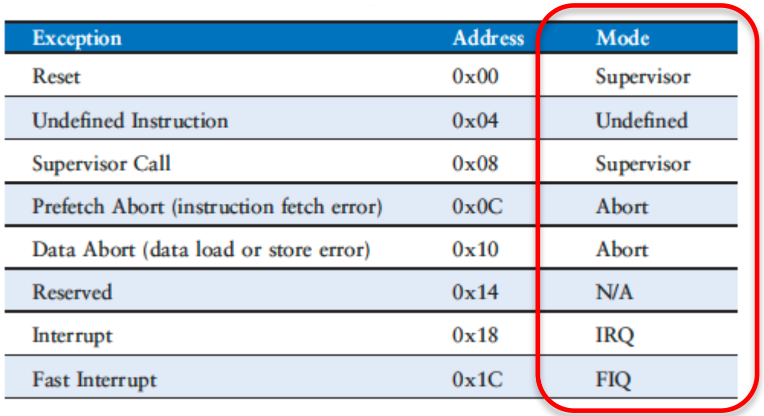
- **Cause**
  - Software exceptions: internal
  - Hardware exceptions: external

- **When to handle**
  - **Internal:** When detected
  - **External:** when convenient
    - Except for very high priority ones or **non-maskable** ones
      - Power failure

- **What if multiple interrupts are raised at the same time?**
  - User can define the priority of an interrupt
  - Interrupt classes

Exception Handling

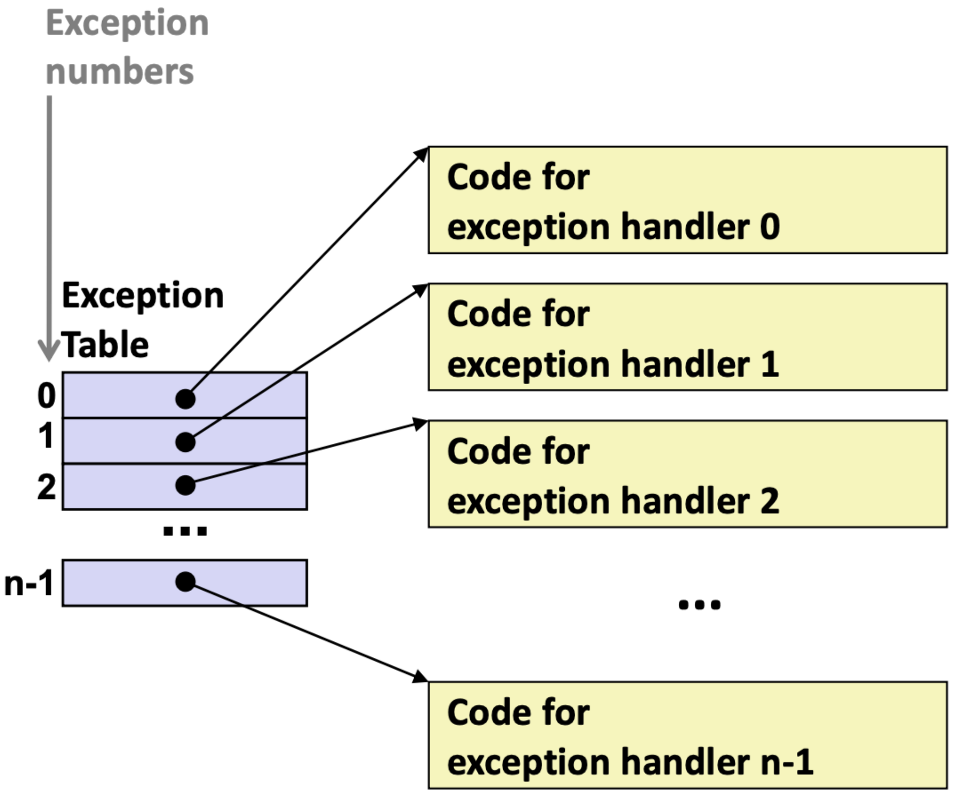
- **Exception handler:** Exceptions use a vector table to determine where to jump to the exception handler



Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

- The table is placed in low memory (recall the memory map)
- Instructions to handle an interrupt is at 0x00000018
- On power-up, CPU goes to address 0x00000000
- The exception vector contains a **branch instruction to an exception handler**, code that handles the exception and then **returns to user code**

Exception Handling


Exception numbers
Exception Table
0
1
2
...
n-1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
...
Code for exception handler n-1

- **Each type of event has a unique exception number (k)**
- **k** = index into the exception table
- Handler **k** is called each time an exception k occurs

ARM Execution Modes

- ARM processors can operate in one of several execution modes with different privilege levels

- The mode is specified in the bottom bits of the CPSR

- User mode operates at privilege level `PL0` and other modes operate at `PL1`, which can access all system resources

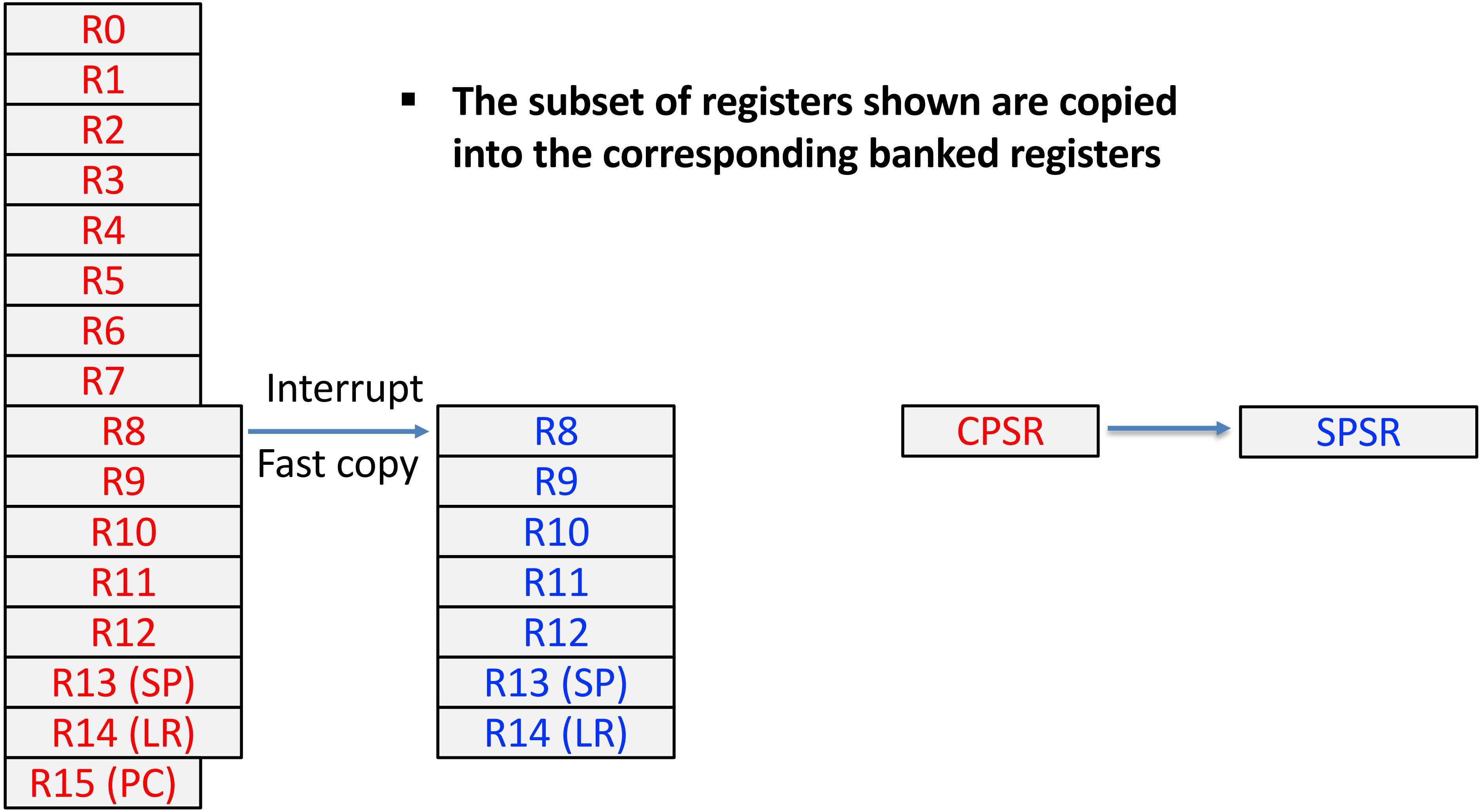
- Execution mode helps the CPU with proper book-keeping
  - Which registers to save (more on this later)
  - Where to return after an interrupt?

Banked Registers

- Exception handler is like a normal function call
  - Need to know where to return
  - Need to preserve registers
- Stack is in memory and memory is slow
- Banked registers are extra (**shadow**) registers that are used to copy values from actual registers on an interrupt
- Idea is to handle interrupts as fast as possible

Example: Fast Interrupt Ex. Mode (FIQ)

- **The subset of registers shown are copied into the corresponding banked registers**



Exception Handling (1)

- Store the CPSR into banked SPSR

- Set the execution mode and privilege level based on the type of exception

- Set the interrupt mask bits in CPSR so that the exception handler will not be interrupted (only for **maskable** interrupts)

- Store the return address into banked `LR`

- Branch to the exception vector table based on exception type

Exception Handling (2)

- Handler does the following
  - pushes other registers onto its stack
  - takes care of exception
  - pops the registers back off the stack

Exception Handling (3)

- The exception handler returns using `MOVS PC, LR`
  - Copies the banked `SPSR` to the `CPSR` to **restore** the status register
  - Copies the banked `LR` to the `PC` to return to the program where the exception occurred
  - Restore the execution mode and privilege level

Transitioning between Privilege Levels

- User code operates at a low privilege level
- Supervisor instruction (`SVC`) is used to transition between levels
- CPU reads the arguments from "certain" registers and executes the specific flavor of the `SVC` instruction
- SVC is a software exception
- A tables specifies **what actions to take** next based on the service user code wants from the OS

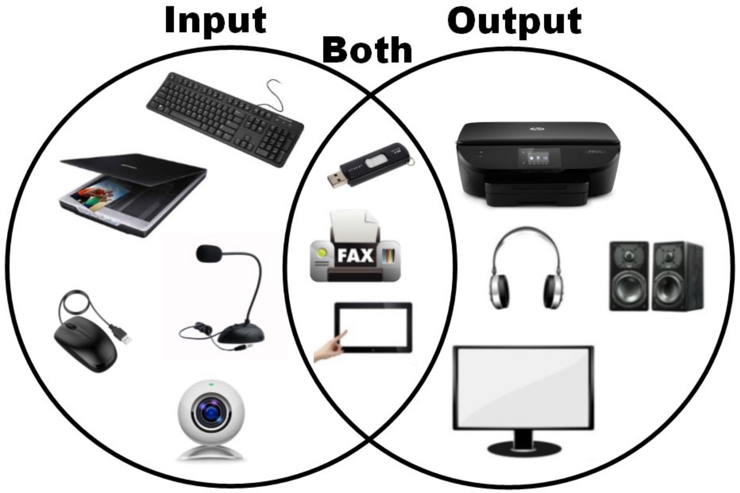
Supervisor Instruction

- Every **“industrial strength”** architecture provides a means to switch between user code and OS code
  - `SVC` in ARM
  - `syscall` in Intel x86
- In the labs, your code is running on “bare metal” – there is no OS
- In real world, only way for user code to access system resources is by invoking functions in the OS (OS is **“trustworthy”** service provider)
  - These functions are called **system calls**

Input/Output (I/O) Architectures

Input/Output Devices

- I/O devices are what makes a system useful
  - Input devices
  - Output devices
  - Storage devices



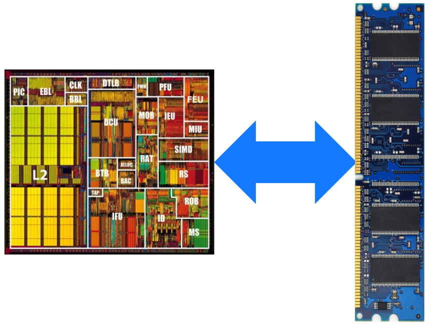
Storage Devices

- Storage devices two important purposes
  - **Persistent storage for long-term storage**
    - Contrast with SRAM/DRAM – Volatile storage
  - **Extension of main memory or DRAM**
    - **Recall:** Memory-resident stack acts as an extension of RF
    - Disk for memory expansion is subject of COMP2310: virtual memory
- Storage devices are very slow compared to DRAM
  - Access latency is in **microseconds** compared to **nanoseconds**
  - For high-end systems used in cloud and datacenters, storage is the most important I/O device

I/O subsystem

- What makes up a computer system?

- CPU-Memory (also called host system)
  - Tightly integrated
  - CPU has **fast/direct** access to memory through address/data buses

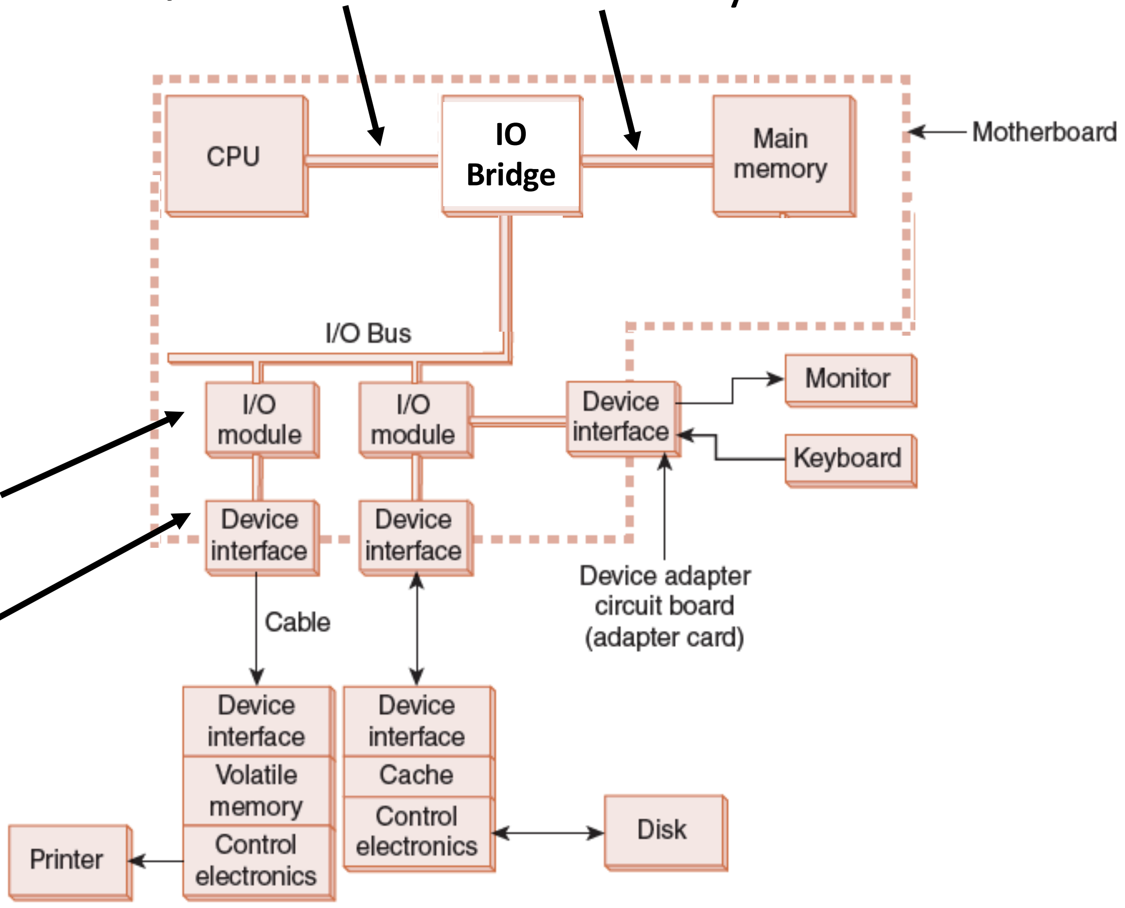


- I/O subsystem
  - Many I/O devices need to communicate to CPU
  - Need a **“subsystem”** to interface with the host system
  - Consisting of devices, buses, communication protocols, data memory

A Model I/O Configuration

System/host bus

Memory bus



Moves data b/w host and a device interface (a.k.a. controller)

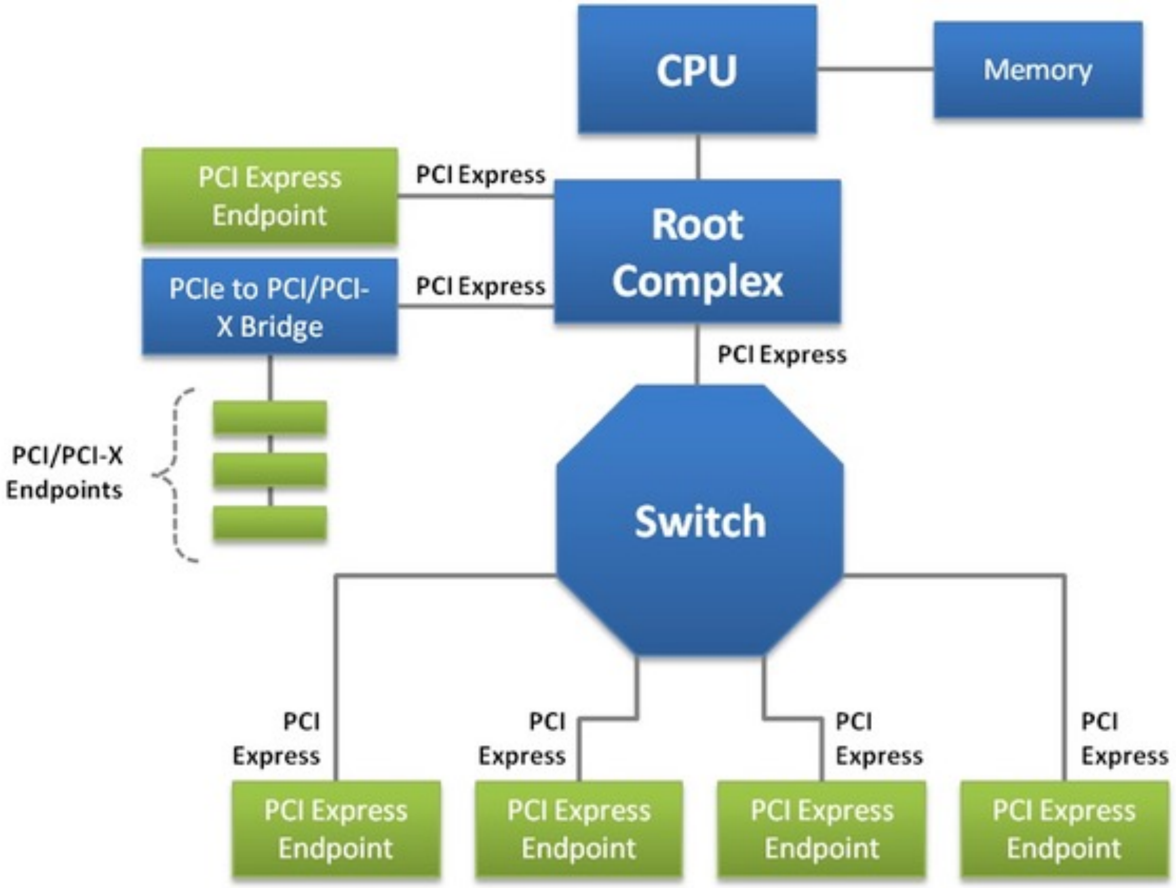
One interface per device class, e.g., USB interface

Modern I/O Bus - PCIe

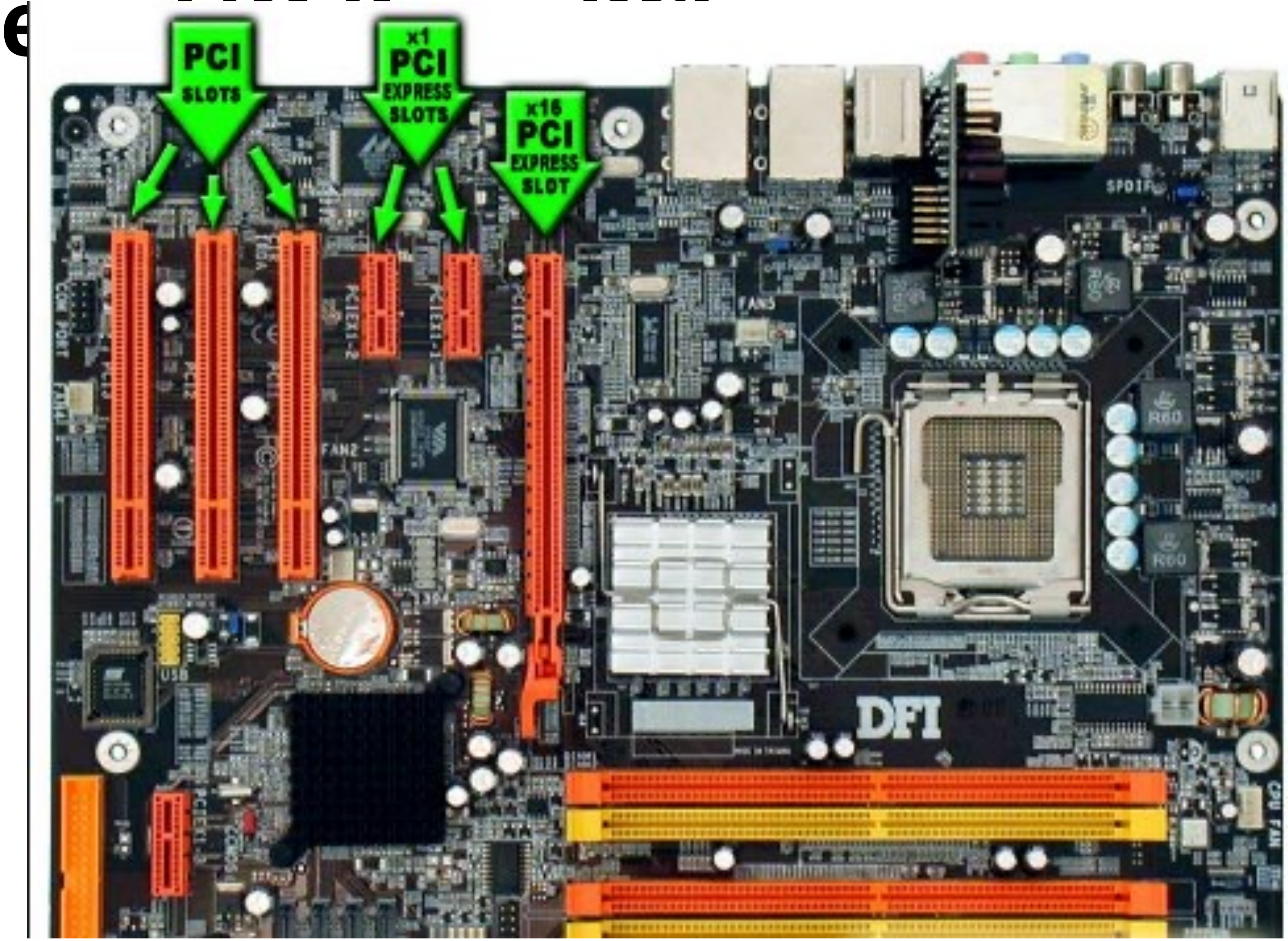
- Peripheral Component Interconnect Express (PCIe or PCI-E) is a **serial expansion bus** standard for connecting a computer to one or more peripheral devices

- PCIe provides lower latency and higher data transfer rates than parallel buses, such as “legacy” PCI

Modern I/O Bus - PCIe



Mode



I/O subsystem

- I/O subsystem
  - Blocks of main memory devoted to I/O functions
  - Buses that provide the means of moving data into and out of the system
  - Control modules in the host and peripheral devices
  - Interfaces to external components such as keyboards and disks
  - Cabling and communications links between the host system and its peripherals

Interfaces and Protocols

- Interfaces communicate with certain types of devices, such as keyboards, disks, or printers

- Interfaces make sure
  - Device is ready for next batch of data
  - Host is ready to receive the next batch of data coming in from the peripheral device

- The exact form and meaning of the signals exchanged between the sender and the receiver is called a protocol
  - Signals are of two types: command and data signals
  - **Handshake:** A protocol exchange in which the receiver sends an Acknowledgement for the commands and data sent or indicate that it is ready to receive data

I/O System Classification

- No standard classification scheme and not every “configuration” makes sense

- Three aspects to classify the systems
  - Device visibility
  - Reading data
  - Event notification

Device Visibility

- We need to somehow make the device visible to the CPU
- Two options
  - Port-mapped or Isolated I/O or instruction-based I/O
  - Memory-mapped I/O

Port-Mapped I/O (PMIO)

- The device is accessible in a dedicated address space, separate from the address space of memory

- I/O devices have a separate address space from general memory, typically accomplished by **extra "I/O" pins** on the CPU's physical interface

- Because the address space for I/O is **isolated** from that for main memory, this is sometimes referred to as isolated I/O

- Special **“dedicated instructions”** to access the I/O address space
  - `IN`
  - `OUT`

Memory-Mapped I/O (MMIO)

- I/O devices and memory share the **same** address space
- Each I/O device has its own **reserved** block of memory
- Data transfers to and from the I/O device involve moving bytes to and from the memory address that is mapped to the device
- MMIO is like using **regular load/store instructions** from the programmer's perspective
- Good abstraction
- Simplicity and convenience (**Yes** for the programmer)

Example MMIO System

- Each I/O device is assigned one or more addresses in the address space

- Good abstraction
- Good architecture
- Neat hardware

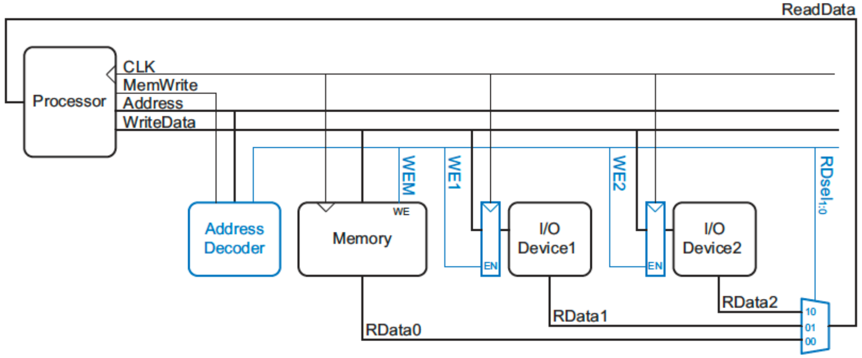


Figure e9.1 Support hardware for memory-mapped I/O

- Recall the memory map with dedicated addresses for I/O devices

- STR instruction writes data to the device

- LDR instruction reads data from the device

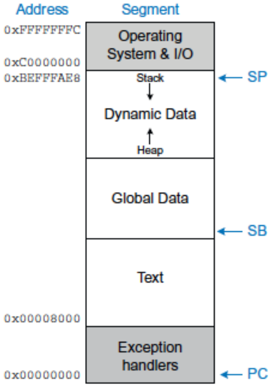


Figure 6.30 Example ARM memory map

Example MMIO System

- Suppose that I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

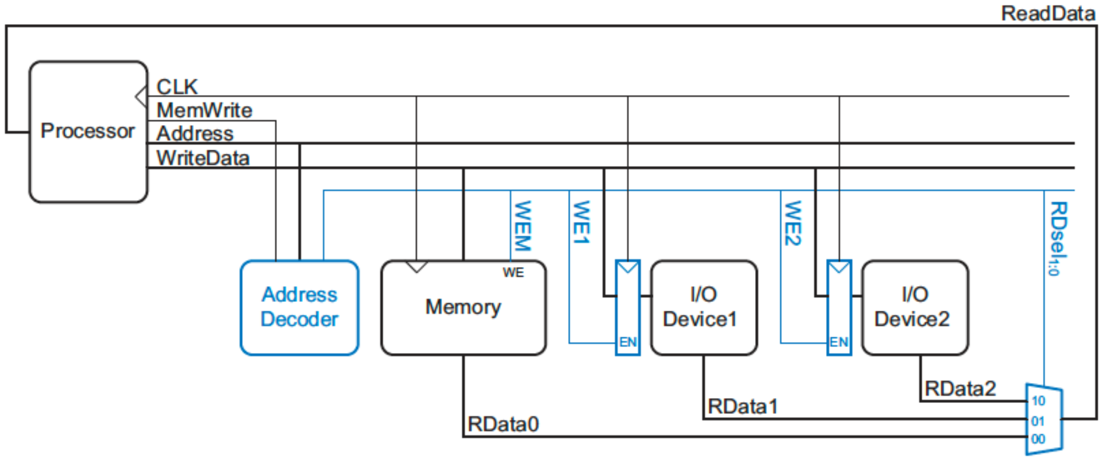


Figure e9.1 Support hardware for memory-mapped I/O

Reading Data

- Programmed I/O

- Direct-Memory Access (DMA)

Programmed I/O (PIO)

- Each data item transfer is initiated by an instruction in the program, involving the CPU for every transaction
  - The term can refer to either memory-mapped I/O (MMIO) or port-mapped I/O (PMIO)
- Why is this a problem?
  - **CPU is very fast (recall: 1 second = 2 billion cycles)**
  - I/O devices are **slow**
  - For large data transfers (for example, reading a video file from disk), we would like to free up the CPU to do other things while transfer happens in parallel

Direct-Memory Access (DMA)

- CPU offloads the execution of tedious I/O instructions to a **dedicated** chip called DMA controller

- CPU provides the DMA controller with
  - the location of the bytes to be transferred
  - the number of bytes to be transferred
  - the destination memory address

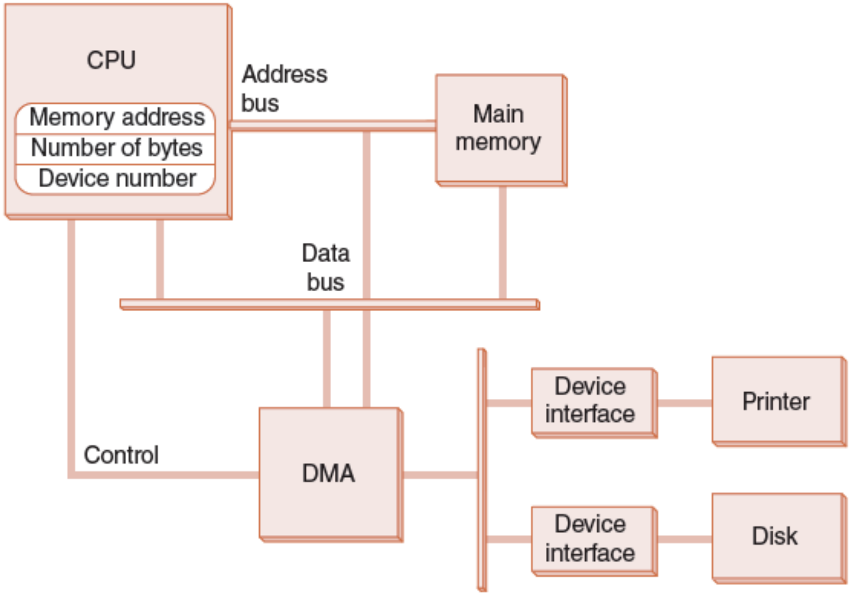
- CPU signals the **DMA controller** and gets busy doing something else

- DMA **takes care** of I/O

- DMA controller places the data in memory and **interrupts** the CPU

A Sample DMA Configuration

- DMA and CPU share the bus (below: memory-mapped I/O)



- DMA runs at a higher priority and steals memory cycles from the CPU

Event Notification

- Polled I/O

- Interrupt-driven I/O

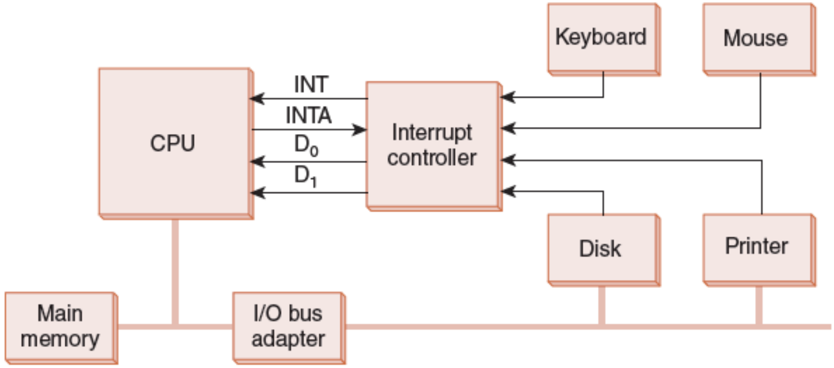
Polled versus Interrupt I/O

- **Polled I/O**
  - CPU monitors a control/status register associated with a port
  - When a byte arrives in the port, a bit in the control register is set
  - The CPU eventually polls and notices that the “data ready” control bit is set
  - The CPU resets the control bit, retrieves the byte, and processes it
  - The CPU resumes polling the register as before

- **Interrupt-driven I/O**
  - CPU is not held up from doing other things
  - Interrupts are asynchronous signals
    - The devices tell the CPU when they have data to send
  - The CPU proceeds with other tasks until a device requesting service sends an interrupt to the CPU
  - **Granularity is configurable:** Interrupts for every word, or for an entire batch

Interrupt-Driven I/O

- Communication between many interrupt-enabled devices and CPU is handled via an interrupt controller



- Once the circuit recognizes an interrupt signal from any device, it raises a single interrupt signal that activates a control line on the system bus
- Control line feeds directly into a pin on the CPU chip
- **INTA:** Interrupt **Ack**nowledge
- INT is lowered by the interrupt controller after receiving the acknowledgement
- Priority is resolved based on the **time-criticality** of the device requesting I/O

Example I/O Systems

- **Memory-mapped DMA interrupt-driven I/O**
  - Typically used for storage devices that transfer large amounts of data

- Port-mapped DMA interrupt driven I/O

- **Port-mapped programmed polling I/O**
  - Polling, programmed, and port-mapped go well together

Character versus Block I/O

- Character I/O devices process one byte (or character) at a time
    - Examples include modems, keyboards, and mice
    - Keyboards are usually connected through an interrupt-driven programmed I/O system

- Block I/O devices handle bytes in groups
    - Most mass storage devices (disk and tape) are block I/O devices
    - Block I/O systems are most efficiently connected through interrupt-driven DMA

- **Device driver:** Software that handles the details of I/O transfer granularity and I/O-related instructions in general

Bus Technology

- Bus is a collection of wires to transfer signals (address, data, control) between sender and receiver

- **Serial transmission:** one bit at a time
  - Keyboard
- **Parallel transmission:** multiple bits in parallel
  - Similar to CPU-Memory communication

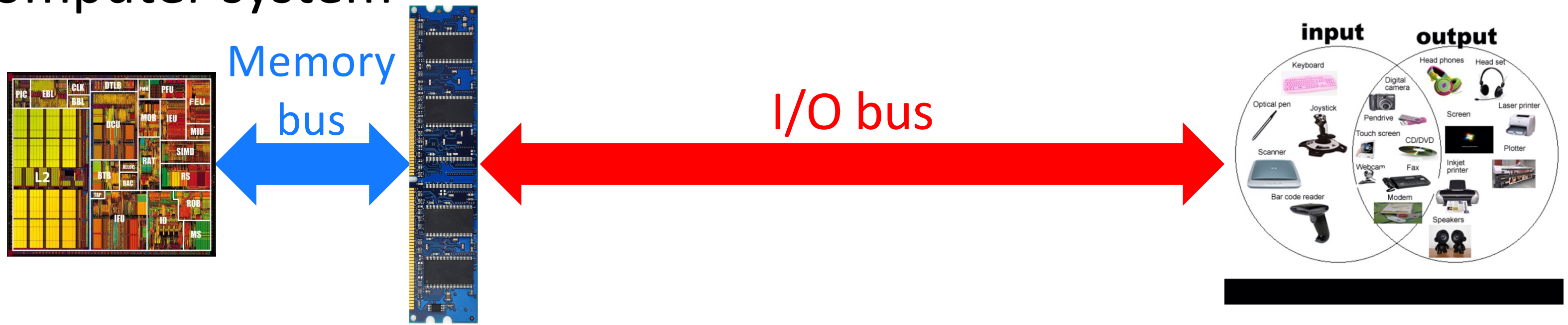
- Parallel cables are fatter than serial cables
  - and susceptible to electrical interference that reduces signal range

- In both cases a clock is used for timing purposes especially controlling signal transitions

Amdahl's Law

I/O and Performance

- Recall the computer system



- Sluggish I/O throughput can have a ripple effect, dragging down overall system performance
- The fastest processor in the world is of little use if it spends most of its time waiting for data from a peripheral device
- If we really understand what’s happening in a computer system, we can make the best possible use of its resources

Amdahl's Law (1)

- The overall performance of a system is a result of the interaction of all of its components

- System performance **is most effectively improved** when the performance of the **most heavily used components** is improved

- This idea is quantified by Amdahl's Law: $S = \frac{1}{(1-f) + (\frac{f}{k})}$
  - *S* is the overall speedup
  - *f* is the fraction of work performed by a **faster** component
  - *k* is the speedup of the faster component

Amdahl’s Law (2)

- Amdahl’s Law gives us a handy way to estimate the performance improvement we can expect when we upgrade a system component

- On a large system, suppose
  - Upgrade a CPU to make it 50% faster for $10,000 **OR**
  - Upgrade its disk drives for $7,000 to make them 150% faster
  - Programs spend 70% of their time running in the CPU **AND** 30% of their time waiting for disk service

- **Question: An upgrade of which component** would offer the **greater benefit** for the lesser cost?

Amdahl’s Law (3)

- Many different way to discuss the notion of “speed-up”

- **In Amdahl’s terms:** 50% **faster means** 1.5 times as fast as (*100% of the speed of the reference object plus an additional 50% of the speed of the reference object – 150% or 1.5 times as fast*)

- Translating speed-up to percentage terminology for use in Amdahl’s law
  - A is N% faster than B if
  - $\frac{time\ B}{time\ A} = 1 + \frac{N}{100}$

Amdahl’s Law (4)

- The processor option offers a 30% speedup:

$$f = 0.70,\ k = 1.5, \text{ so } S = \frac{1}{(1-0.7)+(\frac{0.7}{1.5})} = 1.30$$

- And the disk drive option gives a 22% speedup:

$$f = 0.30,\ k = 2.5, \text{ so } S = \frac{1}{(1-0.3)+(\frac{0.3}{2.5})} = 1.22$$

- Each 1% of improvement for the processor costs \$333, and for the disk a 1% improvement costs \$318

- Should price/performance be your only concern?

Power and Energy

Power and Energy

- Both clock rate and power **increased rapidly** for decades, and the flattened or drop off recently

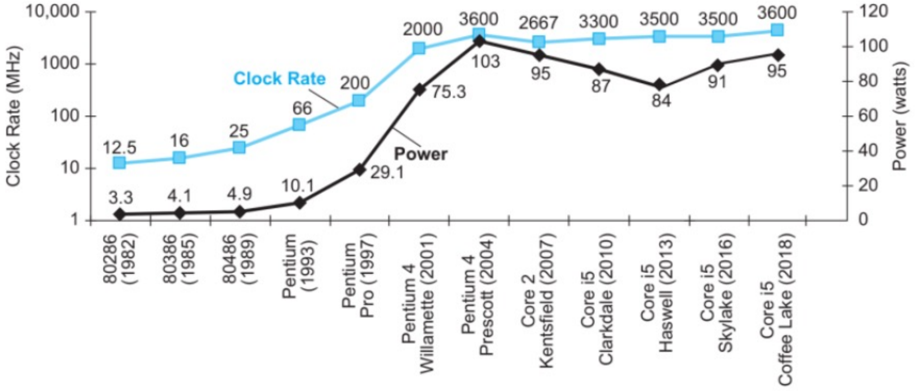


FIGURE 1.16 Clock rate and Power for Intel x86 microprocessors over nine generations and 36 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip. The Core i5 pipelines follow in its footsteps.

- We have run into practical power limits for cooling
- We want to understand the correlation between power and clock frequency

Power and Energy Equations

$$P_{total} = P_{dynamic} + P_{static}$$

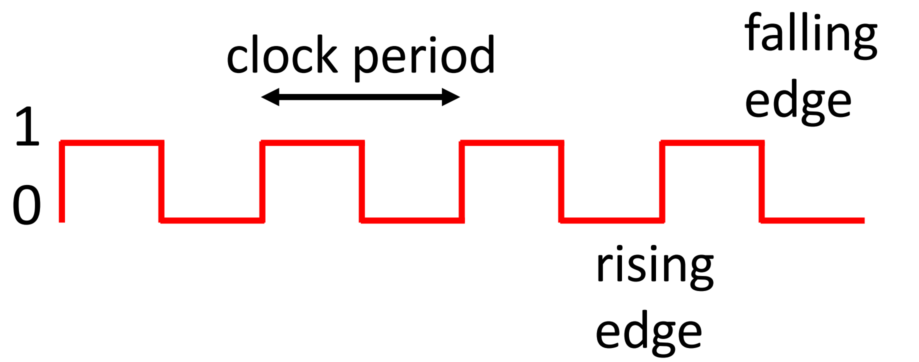
# transistors

$$P_{dynamic} = 1/2 \times A \times C \times V^2 \times N \times F_{switch}$$

$$P_{static} = V \times I_{leak}$$

$$E_{total} = P_{total} \times Time$$

A is activity factor and quantifies how often transistor switches



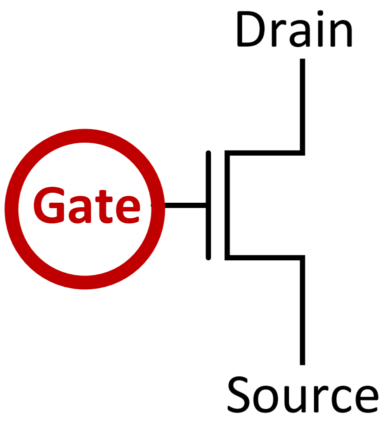
Transistors dissipate power during a transition from **LOW (0)** to **HIGH (1)** and **HIGH (1)** to **LOW (0)** if switching in a cycle

Recall: How Does a Transistor work?

- Instead of the wall switch, we could use an n-type of a p-type MOS transistor to make or break the closed circuit

If the gate of the n-type transistor is supplied with a **high** voltage, the connection from source to drain acts like a piece of wire (**we have a closed circuit**)

If the gate of the n-type transistor is supplied with **zero** voltage, the connection between source and drain is broken (**we have an open circuit**)



Schematic of an n-type MOS transistor

- Depending on the technology, high voltage can range from **0.3V** to **3V**

Power Equation

- $P_{dynamic} = 1/2 \times A \times C \times V^2 \times N \times F_{switch}$
  - F_{switch} depends on the clock rate
  - C is a function of the CMOS technology and fanout: # transistors connected to output of a transistor
  - A and F_{switch} and N kept increasing, leading to the **power wall**
  - Measured in watts, typically reported as peak or average

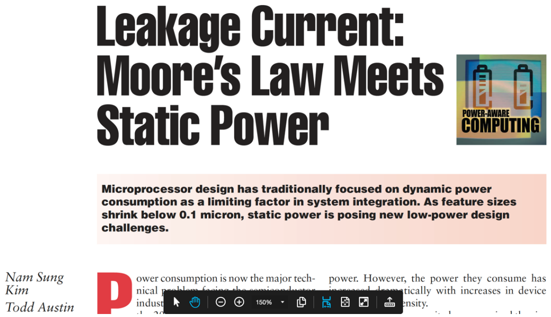
Reducing Dynamic Power

- $P_{dynamic} = 1/2 \times A \times C \times V^2 \times N \times F_{switch}$

- To minimize dynamic power
  - Reduce frequency
  - Reduce supply voltage (**squared reduction**)
- In ~30 years
  - Frequency increased by 1000X
  - Voltage decreased by 15% per generation (from 5V to 1V)
  - Power **increased** by 30X
- Lowering supply voltage is no longer feasible: increases leakage and leads to manufacturing complexity ($)

Dynamic versus Static Power

- **Dynamic power**
  - Primary source of energy consumption
  - Dissipated when the transistor switches
  - Some instruction mixes increase
    - switching activity by flipping control bits



Leakage Current: Moore's Law Meets Static Power

Microprocessor design has traditionally focused on dynamic power consumption as a limiting factor in system integration. As feature sizes shrink below 0.1 micron, static power is posing new low-power design challenges.

Nam Sung Kim
Todd Austin

Power consumption is now the major technical indust

power. However, the power they consume has increased dramatically with increases in device ensity.

- **Static power**
  - <span style="color:red">Due to leakage current that flows even when the transistor is not switching</span>
  - It increased significantly in recent times (40% of total)
  - Further reducing supply voltage increases leakage
    - A phenomenon called thermal runaway
    - Check the paper from **Nam and Todd** if interested

Energy

- Energy is the power dissipated over time
- $E_{total} = (P_{dynamic} + P_{static}) \times \text{time}$
- P_{static} is always there (regardless of activity)
- Measured in joules

Power vs. Energy (with analogies)

In the case of driving a car, power is the rate at which you drove (in miles per hour), and energy is the total distance you drove (in miles).

In the case of a hose with running water going into a bucket, power is the flow rate of the water (liters/second), and energy is how much water ends up in the bucket (liters).

Which metric to use?

- Power: Determines the packaging and cooling requirements
- Energy: To compare the efficiency of two processors

- **Discussion**
  - If maximizing battery life is the goal, it is better to use the energy metric

  - **Think:** Power is a rate metric (joules per second) like MIPS

  - **Recall:** To quantify performance, we use execution time (seconds) and not IPS (instructions per second)

Which CPU/program is more energy efficient?

Same program, different CPUs

	Power (Watts)	Execution time (s)
Processor A	100 Watts	100 seconds
Processor B	80 Watts	150 seconds

Same CPU, different programs

	Power (Watts)	Execution time (s)
C++ program	100 Watts	75 seconds
Java program	100 Watts	750 seconds

Plan

- We are done with **“Program Execution”**

- Advanced microarchitecture in remaining weeks
  - Multicycle (Section 7.4)
  - Pipelined (Section 7.5)
  - Out of Order (Section 7.7 + Slides)