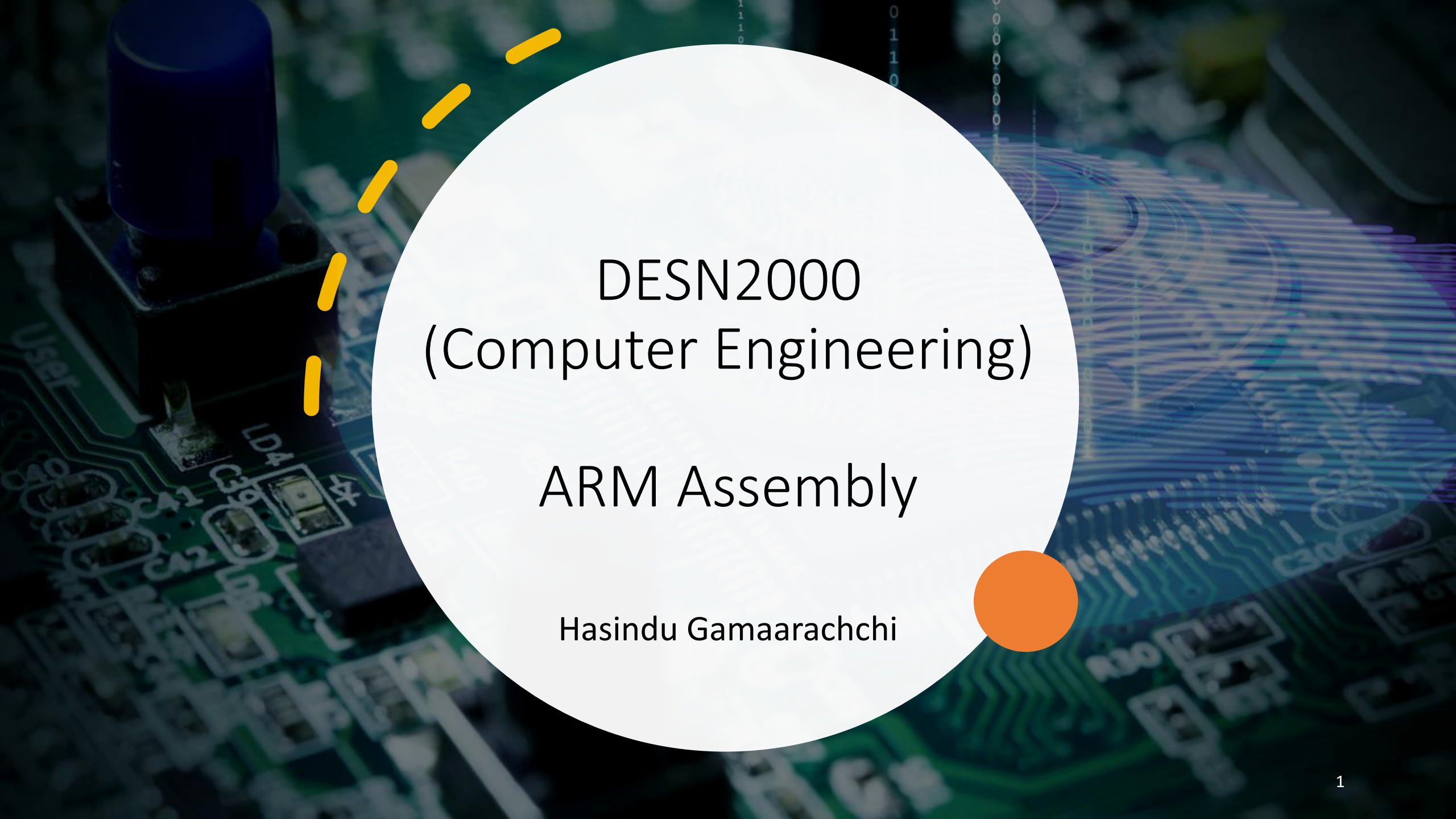


DESN2000 (Computer Engineering)

ARM Assembly

Hasindu Gamaarachchi

ARM Assembly

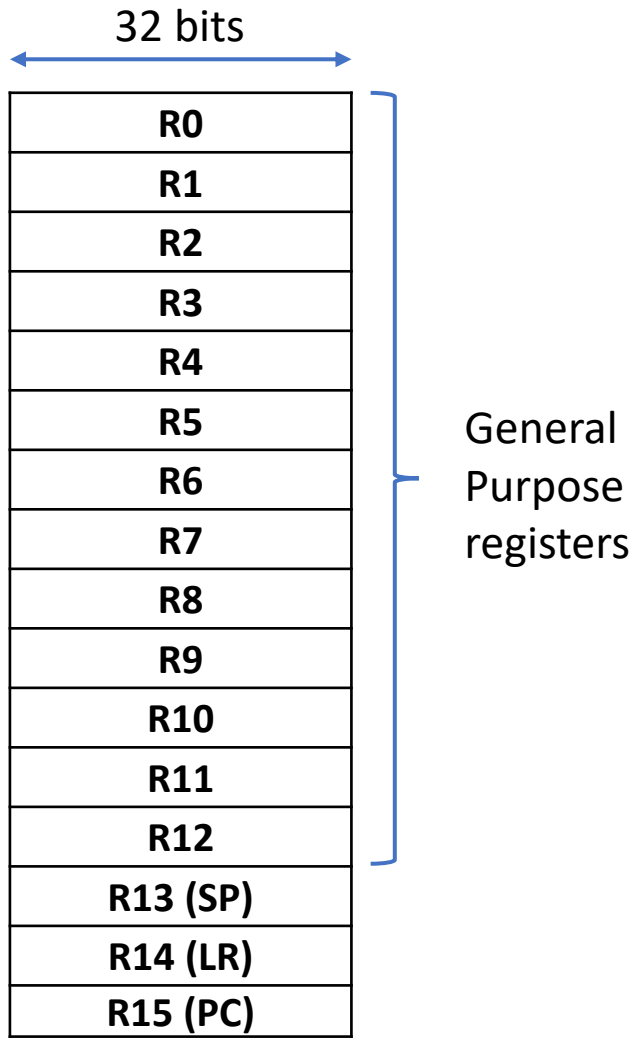
- This is a crash course, well a “crash lecture”!
- Assumes prior learning of an assembly language
  - MIPS in COMP1521
- ARM is a Reduced Instruction Set Computer (RISC) architecture
  - Conceptually a lot of similarities to MIPS
  - And yeh, there are some differences

Learning Resources

The upcoming slides are adapted from “Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C (Fourth Edition)” – Yifeng Zhu

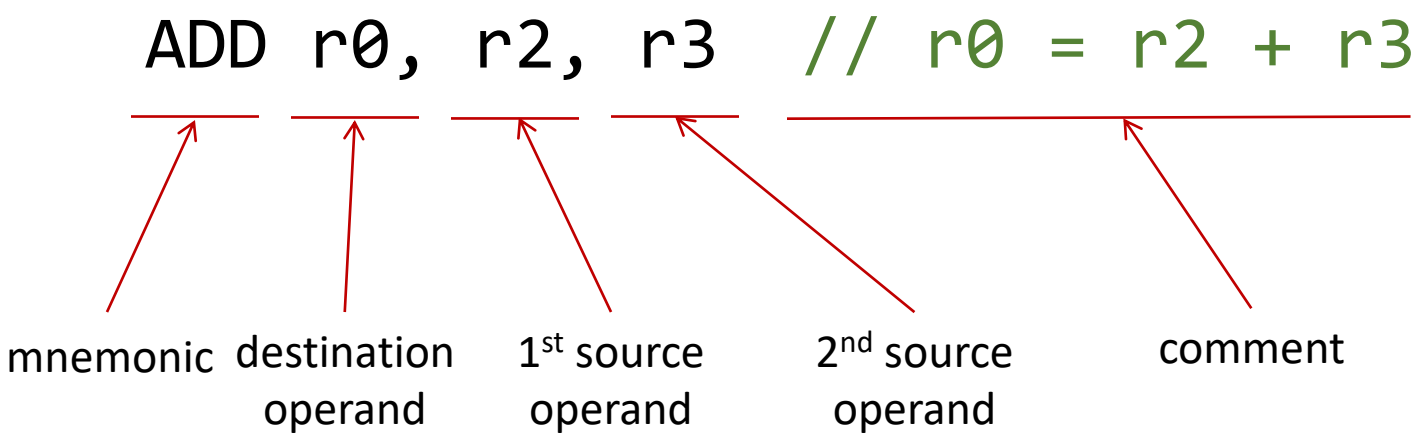
- ARM Instruction set architecture - chapter 3
- For an in-depth understanding of ARM assembly programming – chapters 4-8
- Mixing C and assembly – chapter 10

ARM Processor Register Bank



- Each register is 32 bits
- R0-R12: General purpose registers
- R13: Stack pointer (SP)
- R14: Link Register (LR)
- R15: Programme Counter (PC)

ARM Instruction Examples



ARM Instruction Examples

```
ADD r1, r2, r3      // r1 = r2 + r3
ADD r1, r2, #4      // r1 = r2 + 4

MOV r0, r1          // r0 = r1
MOV r1, #5          // r1 = 5
```

ARM Assembly Instructions

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, If-then, branch,
- Miscellaneous
  - Breakpoints, wait for events, no operation

Arithmetic Instructions: Examples

- `ADD  r0, r1, r2  // r0 = r1 + r2`
- `ADC  r0, r1, r2  // Add with carry, r0 = r1 + r2 + carry`
- `SUB  r0, r1, r2  // r0 = r1 - r2`
- `SBC  r0, r1, r2  // Subtract with borrow, r0 = r1 - r2 – (1 – carry)`
- `MUL  r0, r1, r2  // r0 = r1 * r2, product limited to 32 bits`

Logic Instructions: Examples

- `AND  r0, r1, r2 // Bitwise AND, r0 = r1 AND r2`
- `ORR  r0, r1, r2 // Bitwise OR, r0 = r1 OR r2`
- `EOR  r0, r1, r2 // Bitwise Exclusive OR, r0 = r1 EOR r2`
- `ORN  r0, r1, r2 // Bitwise OR NOT, r0 = r1 ORN r2`

- `BIC  r0, r1, r2 // Bit clear, r0 = r1 & ~r2`

- `LSL  r0, r1, r2 // Logical shift left, r0 =  r1 << r2`
- `LSR  r0, r1, r2 // Logical shift right, r0 =  r1 >> r2`
- `ROR  r0, r1, r2 // Rotate right,  r0 = r1 rotate by r2 bits`

Data Movement Instructions: Examples

```
• MOV   r1, r0       // r1 = r0
• MOV   r1, #16      // r1 = 16
• LDR   r1, =10000   // r1 = 10000

// assume r0 has the memory address
• LDR   r1, [r0]     // r1 = Memory.word[r0]
• LDR   r1, [r0,#8]  // r1 = Memory.word[r0+8]

; assume r0 has the memory address
• STR   r1, [r0]     // Memory.word[r0] = r1
• STR   r1, [r0,#8]  // Memory.word[r0+8] = r1
```

Compare and branch Instructions: Examples

C Program

```
if (a == 1)
    b = 3;
else
    b = 4;
```

If-then-else

```
        // r1 = a, r2 = b
        CMP r1, #1    // compare a and 1
        BNE else      // go to else if a ≠ 1
then:   MOV r2, #3    // b = 3
        B   endif     // go to endif
else:   MOV r2, #4    // b = 4
Endif:
```

Compare and branch Instructions: Examples

C Program

```
int i;
int sum = 0;
for(i = 0; i < 10; i++){
  sum += i;
}
```

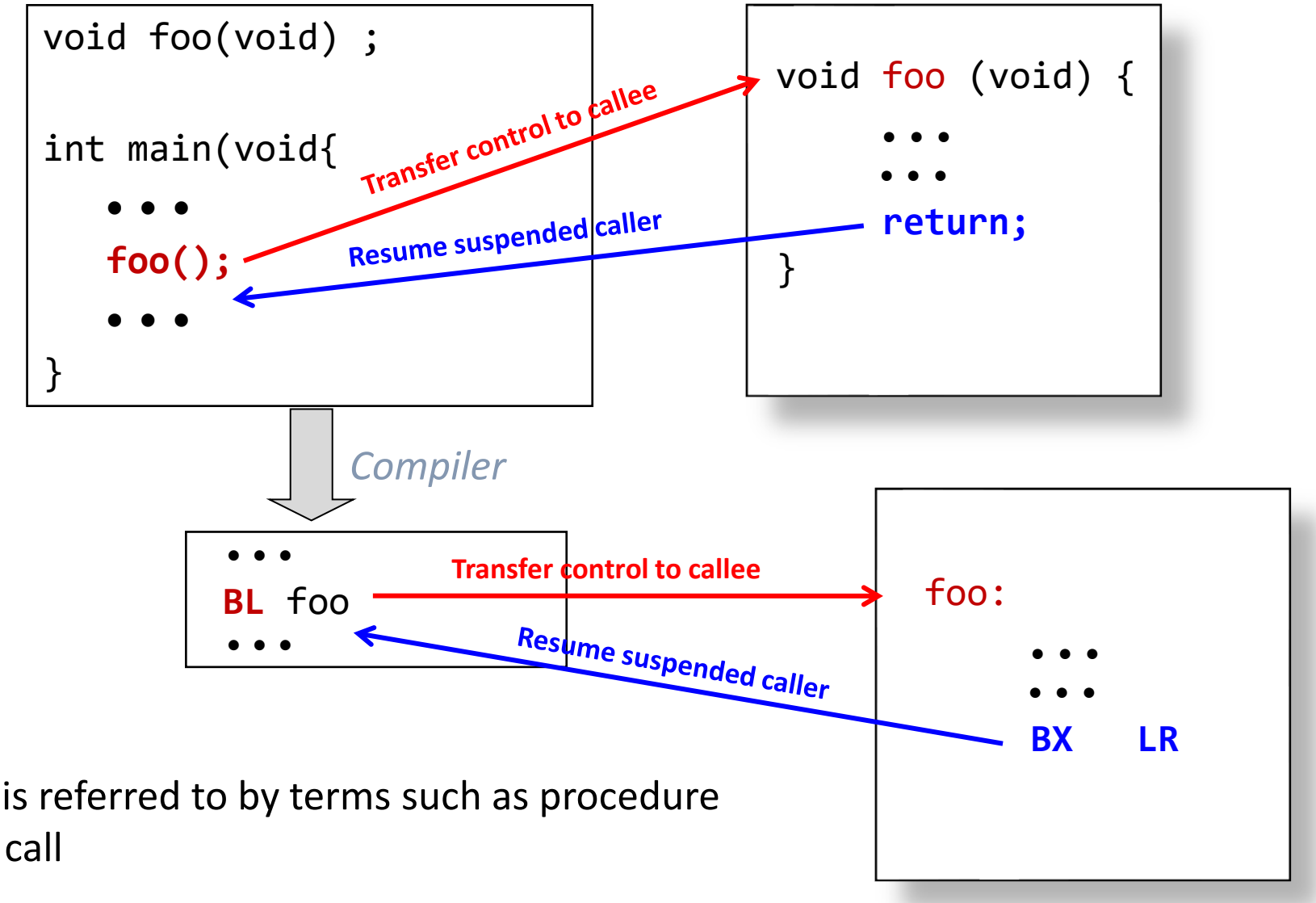
For Loop

```
        MOV r0, #0   // i
        MOV r1, #0   // sum

loop:   CMP r0, #10      // compare i and 10
        BGE   stop       // end loop if i>=10
        ADD r1, r1, r0   // sum += i
        ADD r0, r0, #1   // i++
        B loop           // loop
stop:
```

Compare and branch Instructions: Examples

Function Call



Note: function call is referred to by terms such as procedure call and subroutine call

Miscellaneous Instructions: Examples

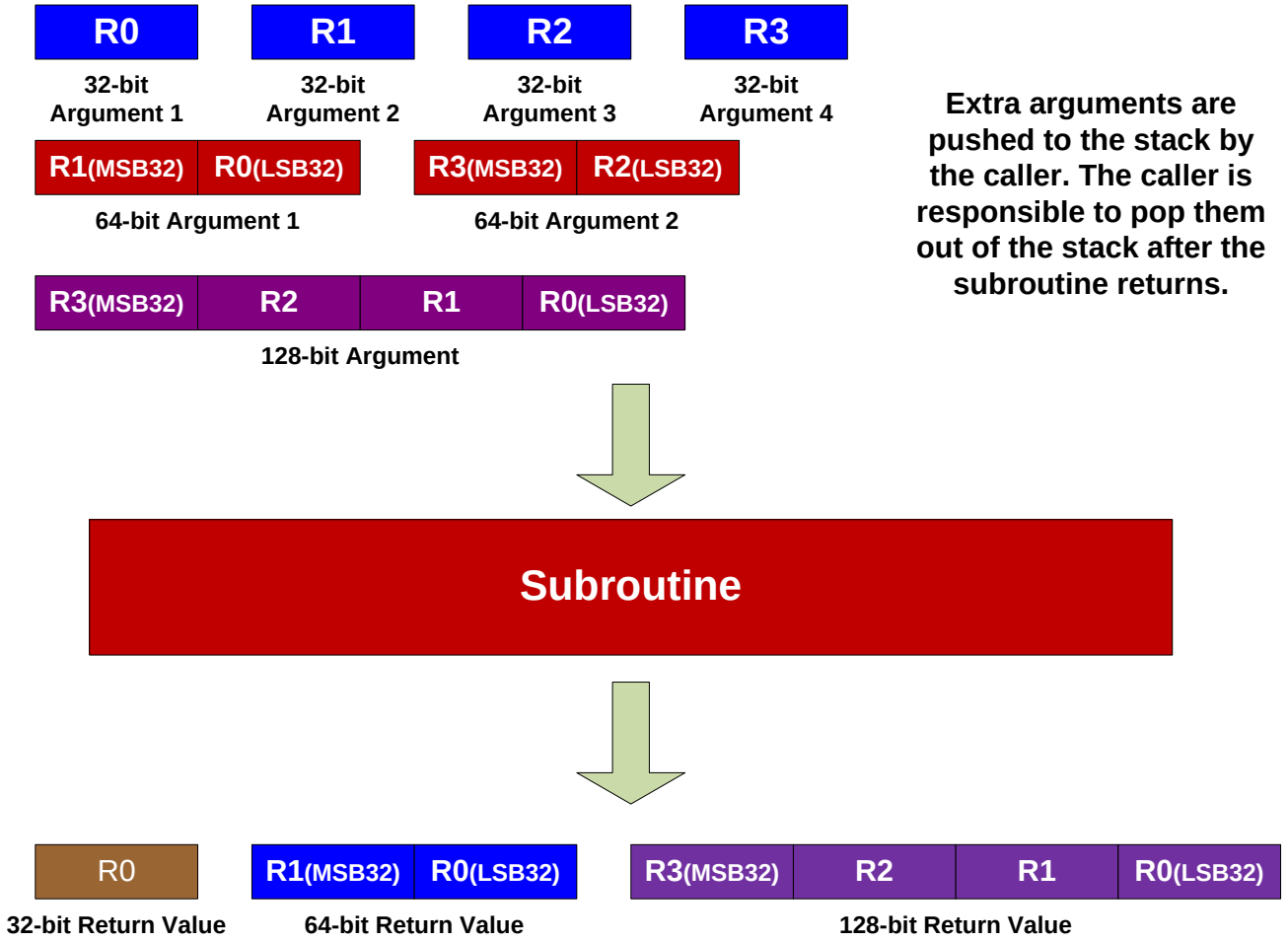
- **NOP** `// No Operation`

Function Calls – Calling Standard

- What is it?
  - Contract between a caller and callee
- Why need it?
  - Allows functions to be separately written, compiled, and assembled but work together
  - Allows C program call an assembly function, or vice versa
- Involves:
  - Argument passing
  - Return values
  - Backing up registers

- Calling standard is also known by terms like calling convention

Passing Arguments and Returning Value



Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

Passing Arguments and Returning Value

```
int32_t sum(uint8_t a8, int8_t b8, uint16_t c16, uint16_t d16);

s = sum(1, 2, 3, 4);
```

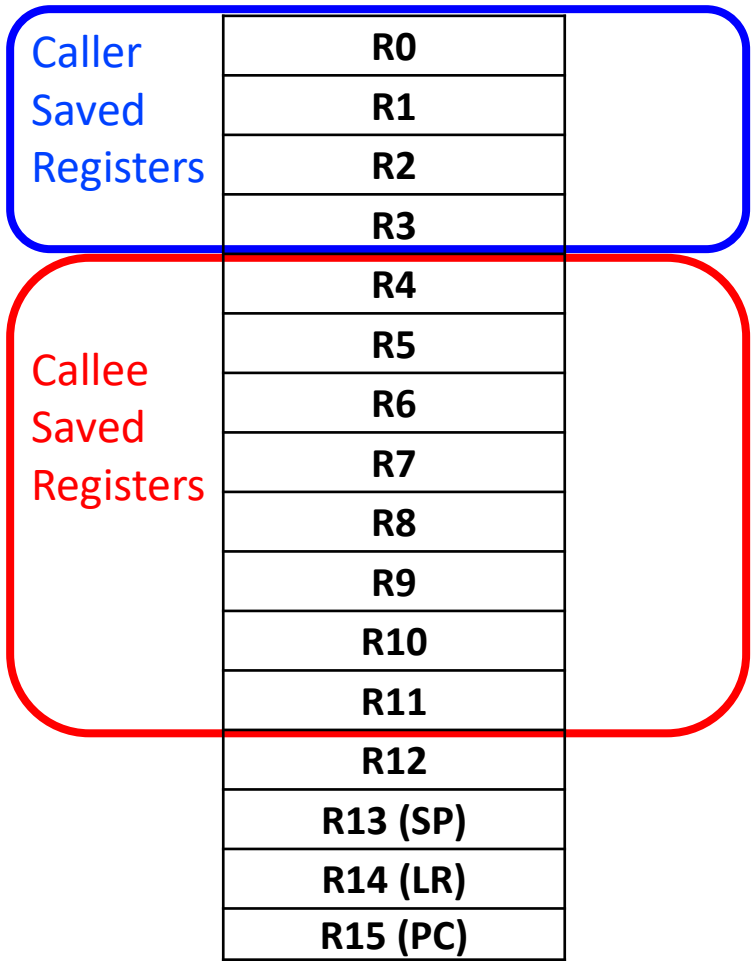
Caller

```
  MOVS r0, #1   // a8
  MOVS r1, #2   // b8
  MOVS r2, #3   // c16
  MOVS r3, #4   // d16
  BL   sum
```

Callee

```
Sum:
  ADD r0, r0, r1   // a8 + b8
  ADD r0, r0, r2   // add c16
  ADD r0, r0, r3   // add d16
  BX  LR
```

Preserving registers



- **Callee can freely modify R0, R1, R2, and R3**
- **If caller expects their values are retained, caller should push them onto the stack before calling the callee**

- **Caller expects these values are retained .**
- **If Callee modifies them, callee must restore their values upon leaving the function.**

Preserving registers

Caller Program	Subroutine/Callee

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1 // r4 = 101, not 11
	foo:
	PUSH {r4} // preserve r4
	...
	MOV r4, #10 // foo changes r4
	...
	POP {r4} // Recover r4
	BX LR

Caller expects callee does not modify r4!

Callee should preserve r4!

Mixing C and Assembly

- Inline assembly

```
// ...
// C code
__asm__(
"MOV r0, #0x048000000 \n"
"LDR r1, [r0, #0x14] \n"
"BIC r1, r1, #1<<5 \n"
"STR r1, [r0, #0x14] \n"
);
//...
// C code
```

Mixing C and Assembly

- Calling as assembly function from C

C Program (main.c)

```
char str[25] = "Hello!";

extern void int strlen(char* s);

int main(void){
    int i;
    i = strlen(str);
    while(1);
}
```

Assembly Program (strlen.s)

```
.global strlen
strlen:
        PUSH {r4}           // preserve r4
        MOV  r4, #0         // initialize length
loop:
        LDRB r1, [r0, r4]   // r0 = string address
        CBZ  r1, exit       // branch if zero
        ADD  r4, r4, #1     // length++
        B    loop           // do it again
Exit:
        MOV  r0, r4         // place result in r0
        POP  {r4}           // return
        BX LR
```

Arithmetic and Logic Instructions

- **Shift** : **LSL** (logic shift left), **LSR** (logic shift right), **ASR** (arithmetic shift right), **ROR** (rotate right), **RRX** (rotate right with extend)
- **Logic**: **AND** (bitwise and), **ORR** (bitwise or), **EOR** (bitwise exclusive or), **ORN** (bitwise or not), **MVN** (move not)
- **Bit set/clear**: **BFC** (bit field clear), **BFI** (bit field insert), **BIC** (bit clear), **CLZ** (count leading zeroes)
- **Bit/byte reordering**: **RBIT** (reverse bit order in a word), **REV** (reverse byte order in a word), **REV16** (reverse byte order in each half-word independently), **REVSH** (reverse byte order in each half-word independently)
- **Addition**: **ADD**, **ADC** (add with carry)
- **Subtraction**: **SUB**, **RSB** (reverse subtract), **SBC** (subtract with carry)
- **Multiplication**: **MUL** (multiply), **MLA** (multiply-accumulate), **MLS** (multiply-subtract), **SMULL** (signed long multiply-accumulate), **SMLAL** (signed long multiply-accumulate), **UMULL** (unsigned long multiply-subtract), **UMLAL** (unsigned long multiply-subtract)
- **Division**: **SDIV** (signed), **UDIV** (unsigned)
- **Saturation**: **SSAT** (signed), **USAT** (unsigned)
- **Sign extension**: **SXTB** (signed), **SXTH**, **UXTB**, **UXTH**
- **Bit field extract: SBFX** (signed), **UBFX** (unsigned)

Load/Store a Byte, Halfword, Word

```
LDRxxx R0, [R1]
// Load data from memory into a 32-bit register
```

LDR	Load Word	`uint32_t/int32_t`	`unsigned or signed int`
LDRB	Load **B**yte	`uint8_t`	`unsigned char`
LDRH	Load **H**alfword	`uint16_t`	`unsigned short int`
LDRSB	Load **S**igned **B**yte	`int8_t`	`signed char`
LDRSH	Load **S**igned **H**alfword	`int16_t`	`signed short int`

```
STRxxx R0, [R1]
// Store data extracted from a 32-bit register into memory
```

STR	Store Word	`uint32_t/int32_t`	`unsigned or signed int`
STRB	Store Lower **B**yte	`uint8_t/int8_t`	`unsigned or signed char`
STRH	Store Lower **H**alfword	`uint16_t/int16_t`	`unsigned or signed short`

Pre-index and Post-index

Index Format	Example	Equivalent
Pre-index	LDR r1, [r0, #4]	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	LDR r1, [r0], #4	r1 ← memory[r0] r0 ← r0 + 4

Offset range is -255 to +255

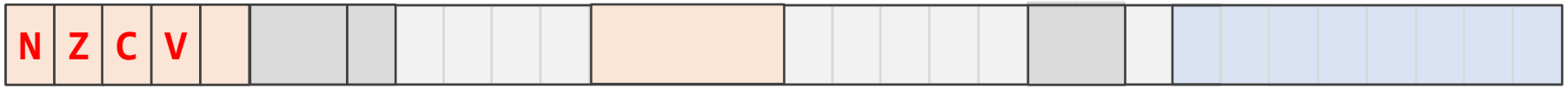
Unconditional Branch Instructions

Instruction	Operands	Brief description
B	`label`	Branch
BL	`label`	Branch with Link
BX	`Rm`	Branch indirect

- *B label*
  - perform a branch to label.
- *BL label*
  - copy the address of the next instruction into `r14` (lr, the link register), and
  - perform a branch to label.
- *BX Rm*
  - branch to the address held in Rm

Condition Flags

Program Status Register (PSR)



- **Negative** bit
  - `N = 1` if most significant bit of result is 1
- **Zero** bit
  - `Z = 1` if all bits of the result are `0`
- **Carry** bit
  - For unsigned addition, `C = 1` if carry takes place
  - For unsigned subtraction, `C = 0` (carry = not borrow) if borrow takes place
  - For shift/rotation, C = last bit shifted out
- **oVerflow** bit
  - `V = 1` if adding 2 same-signed numbers produces a result with the opposite sign
    - Positive + Positive = Negative, or
    - Negative + negative = Positive
  - Non-arithmetic operations does not touch V bit, such as `MOV`,`AND`,`LSL`,`MUL`

Branch Instructions

	Instruction	Description	Flags tested
Conditional Branch	BEQ label	Branch if EQual	Z = 1
	BNE label	Branch if Not Equal	Z = 0
	BCS/BHS label	Branch if unsigned Higher or Same	C = 1
	BCC/BLO label	Branch if unsigned LOwer	C = 0
	BMI label	Branch if MInus (Negative)	N = 1
	BPL label	Branch if PLus (Positive or Zero)	N = 0
	BVS label	Branch if oVerflow Set	V = 1
	BVC label	Branch if oVerflow Clear	V = 0
	BHI label	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS label	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE label	Branch if signed Greater or Equal	N = V
	BLT label	Branch if signed Less Than	N != V
	BGT label	Branch if signed Greater Than	Z = 0 & N = V
	BLE label	Branch if signed Less than or Equal	Z = 1 or N = !V

ARM Procedure Call Standard

Register	Usage	Subroutine Preserved	Notes
r0	Argument **1** and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument **2**	No	
r2	Argument **3**	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument **4**	No	If more than 4 arguments, use the stack
r4	General-purpose V1	Yes	Variable register 1 holds a local variable.
r5	General-purpose V2	Yes	Variable register 2 holds a local variable.
r6	General-purpose V3	Yes	Variable register 3 holds a local variable.
r7	General-purpose V4	Yes	Variable register 4 holds a local variable.
r8	General-purpose V5	YES	Variable register 5 holds a local variable.
r9	Platform specific/V6	Yes/No	Usage is platform-dependent. Can be Variable register 6
r10	General-purpose V7	Yes	Variable register 7 holds a local variable.
r11	General-purpose V8	Yes	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP has to be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC

ARM Cortex Instruction Set

- https://web.eece.maine.edu/~zhu/book/Appendix_B_Cortex_M3_M4_Instructions.pdf

ARM Immediate Values

- Following posts would be helpful those who want to know how and what immediate values are supported in ARM instructions
  - https://xlogicx.net/ARM_12-bit_Immediates_are_Too_High_Level.html
  - https://minhhua.com/arm_immed_encoding/index.html