Computer Organization and Architecture

COSC 2425

Lecture – 9

Sept 19th , 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

Chapter 2

Instructions: Language of the Computer

Other Comparison

<	Less than
≤	Less than or equal
>	Greater than
≥	Greater than or equal
=	Equal
! =	Not equal

```
if ( i > j )
..
SUB X9, i, j
//check if +ve
```

Requires too much logic.

All these conditions can be checked by setting four flags, called *Condition Codes*

Condition code

- LEGv8 provides four added bits called condition codes.
- Some arithmetic instructions can optionally set these flags based on the result of the operation.
- Then the branch (B) instruction can check these bits to do comparisons.

Condition codes/flags

$Negative$(N)	
$Zero$ (Z)	
$Overflow$ (V)	
$Carry$ (C)	

Example SUBS : Subtract and Set Flag

- LEGv8 provides set flag variants for SUB

$Assume\ i =\ +9$, j = +10 are signed integers, and store in X1, and X2 respectively

To do the comparison

$$If\ (i\ <\ j)$$

…

LEGv8 code:

$$SUBS\ X1, X1, X2$$

$$//\ Branch\ if\ N\ flag\ is\ set$$

Condition codes/flags

$Negative$(N)	1
$Zero$ (Z)	
$Overflow$ (V)	
$Carry$ (C)	

Conditional branches use these codes to do comparisons

Four Condition Flags

- negative (N): result had 1 in MSB
- zero (Z): result was 0
- overflow (V): result overflowed
- carry (C): result had carryout from MSB

Set Flag Instructions

Arithmetic Instruction	With Set Flag Option (Suffix S)	Description
ADD	**ADDS**	**Add and set condition flag**
ADDI	**ADDIS**	**Add immediate and set condition flag**
SUB	**SUBS**	**Subtract and set condition flag**
SUBI	**SUBIS**	**Subtract immediate and set condition flag**
AND	**ANDS**	**AND and set condition flag**
ANDI	**ANDIS**	**AND immediate and set condition flag**

Conditional Branches that use Flags

- Format → B.cond
- Use subtract to set flags and then conditionally branch
  - **B.EQ**
  - **B.NE**
  - **B.LT** (less than, **signed**)
  - **B.LO** (less than, unsigned)
  - **B.LE** (less than or equal, **signed**)
  - **B.LS** (less than or equal, unsigned)
  - **B.GT** (greater than, **signed**)
  - **B.HI** (greater than, unsigned)
  - **B.GE** (greater than or equal, **signed**),
  - **B.HS** (greater than or equal, unsigned)

Conditional Example

```
if (a > b)
        a += 1;
```

– a in X22, b in X23

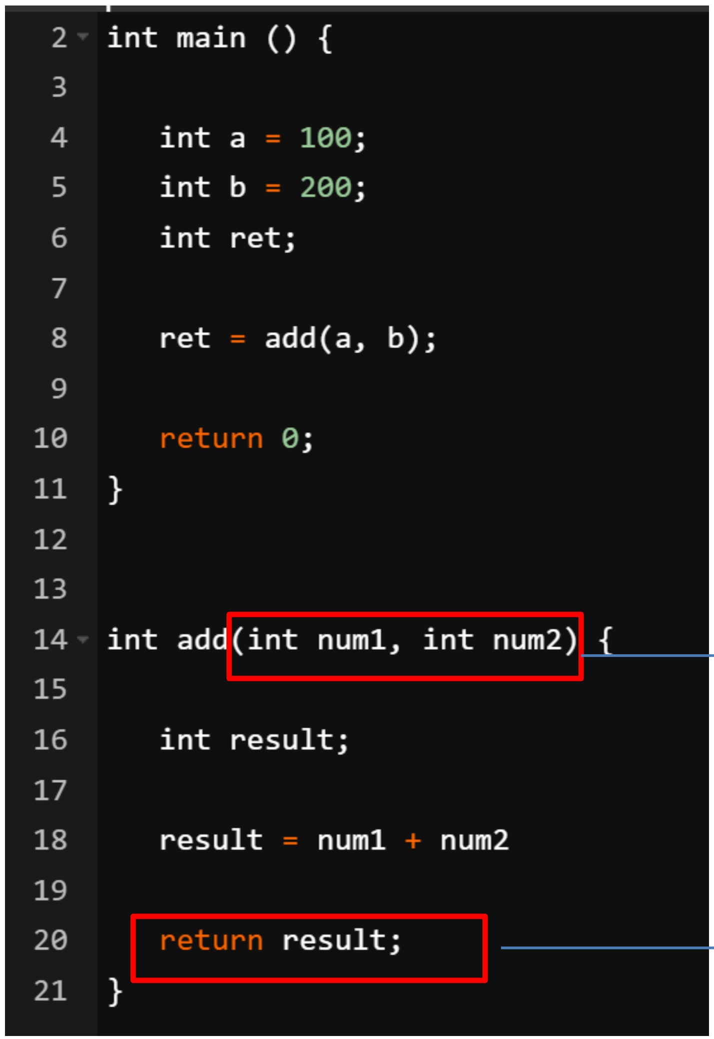
LEGv8 Code:

```
   SUBS X9,X22,X23  // use subtract to make comparison
   B.LTE Exit       // conditional branch
   ADDI X22,X22,#1
Exit:
```

Supporting Procedures in Computer Hardware

- Procedure or functions:
  - Structure programs
  - Easy to read
  - Reusable code

C Example



```c
int main () {

    int a = 100;
    int b = 200;
    int ret;

    ret = add(a, b);

    return 0;
}


int add(int num1, int num2) {

    int result;

    result = num1 + num2

    return result;
}
```

Has parameters

Has return value

Procedure Calling

- Steps required
  1. Place parameters in registers X0 to X7
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call (address in X30)

Procedure Instructions

- Procedure call: jump and link

```
BL ProcedureLabel
```

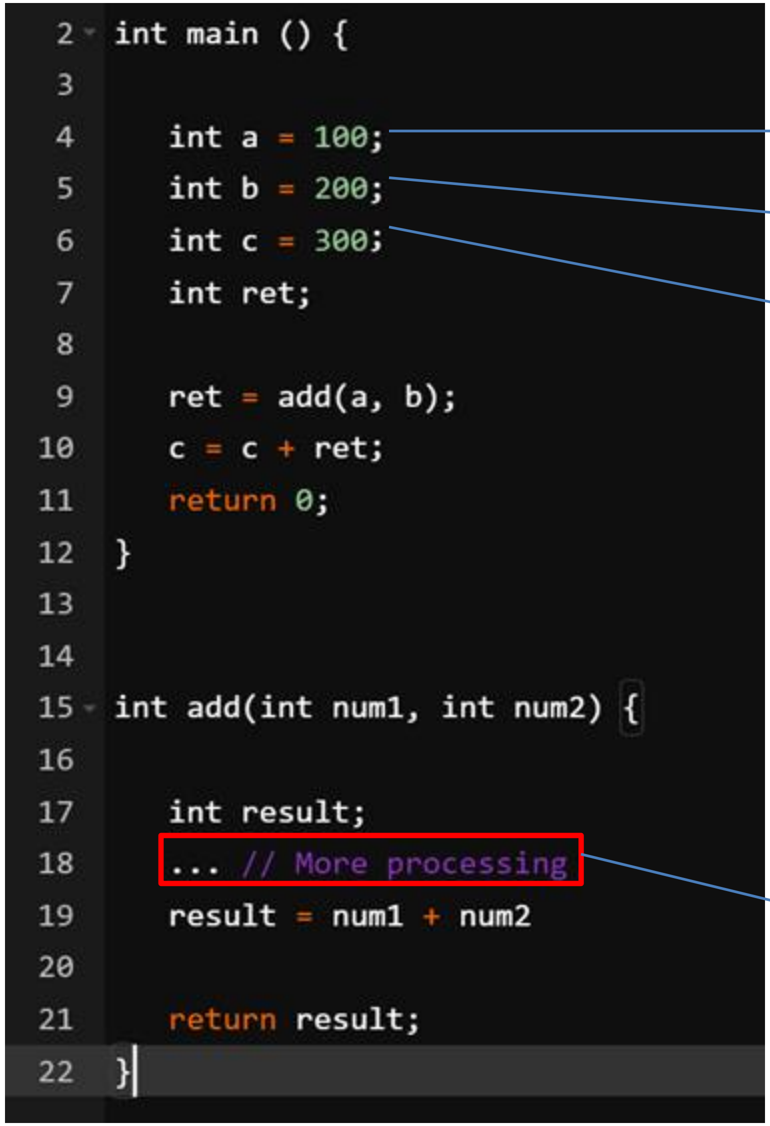
  - BL: Branch and Link Register
  - Address of following instruction put in X30 (LR)
  - Jumps to target address

- Procedure return: jump register

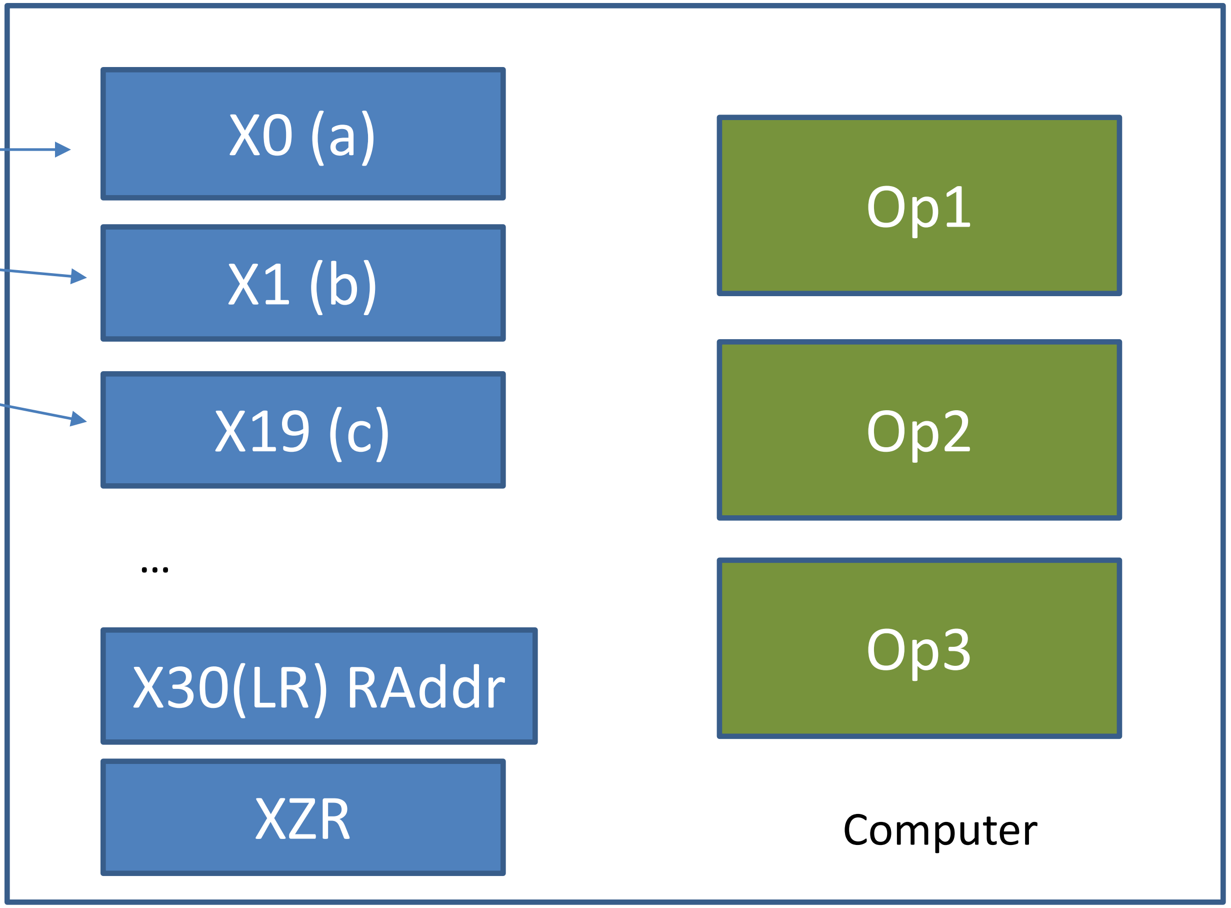
```
BR LR
```

  - BR: Branch Register
  - Copies LR to program counter

What if > 8 registers are needed by Callee?



```
int main () {

    int a = 100;
    int b = 200;
    int c = 300;
    int ret;

    ret = add(a, b);
    c = c + ret;
    return 0;
}


int add(int num1, int num2) {

    int result;
    ... // More processing
    result = num1 + num2

    return result;
}
```



Needs more registers?

Spill and Restore Registers

1. Save variable c to memory from register
    1. A register spill is said to occur
2. Finish executing procedure
3. Restore value of variable c from memory to Previous location (X19)

One register contains memory location to store the values

The ideal structure to store values is a *Stack*

What Registers used?

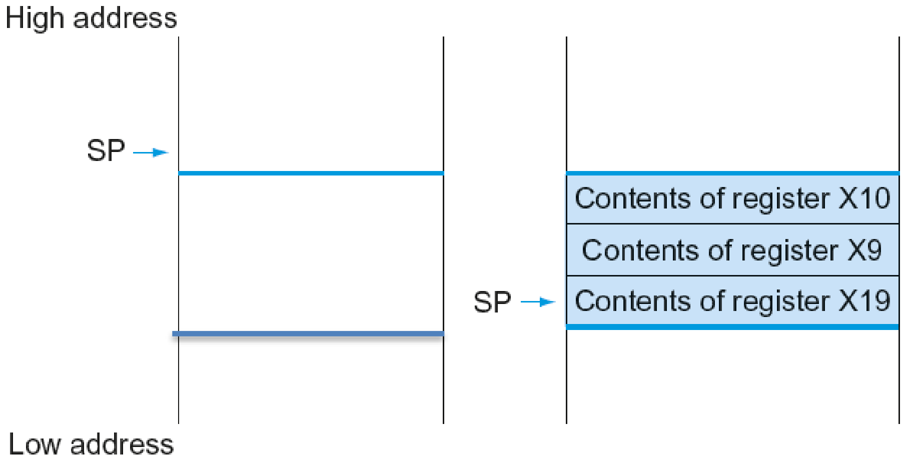
- X0 – X7: procedure arguments/results
- **X28 (SP): stack pointer (address of the most recently allocated stack)**
- X30 (LR): link register (return address)
  - Also called as program counter (PC)

Spilling to stack

- To spill registers (X10, X9, X19) on to the stack
- Address of stack is save in X28 (SP)

LEGv8 Code:

```
SUBI SP, SP, #24 // Make room for three items
STUR X10, [SP, #16]//Spill X10
STUR X9, [SP, #8]//Spill X 9
STUR X19, [SP, #0]//Spill X 19
```



Restore from Stack

- To spill registers (X10, X9, X19) on to the stack
- Address of stack is save in X28 (SP)

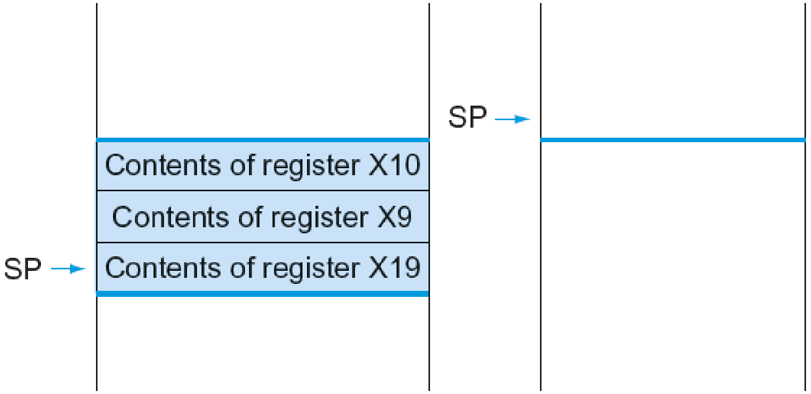
LEGv8 Code:

$LDUR\ X19, [SP, \#0] // Spill\ X\ 19$ ***(POP)***

$LDUR\ X9, [SP, \#8] // Spill\ X\ 9$

$LDUR\ X10, [SP, \#16] // Spill\ X10$

$ADDI\ SP, SP, \#24\ //\ Make\ room\ for\ three\ items$



Procedures

- **Non-Leaf Procedures:** Procedures that make calls to (or invoke) other procedures.
- **Leaf procedures:** Procedures that do not make calls to other procedures

Leaf Procedure Example

- C code:

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

Leaf Procedure Example

- C code:

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

  - *long int – (32 bit integer)*
  - *long long int – (64 bit integer)*

Leaf Procedure Example

- C code:

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

  - Four arguments (g, h, I, j)
  - Arguments g, …, j in X0, …, X3

Leaf Procedure Example

- C code:

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

  - Four arguments (g, h, I, j)
  - Arguments g, …, j in X0, …, X3
  - One return (f)
  - f in X19 (hence, need to save on stack)

Leaf Procedure Example

- C code:

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

  - Four arguments (g, h, I, j)
  - Arguments g, …, j in X0, …, X3
  - One return (f)
  - f in X19

Caller:

1. The caller is using temporaries (X9, X10, X19)

The caller executes branch and link, to pass control to the caller

$BL\ leaf\_example$

Leaf Procedure Example

1. Spill registers

2. Do Procedure

3. Restore registers
Jump to return
address

Leaf Procedure Example

1. Spill registers

Save registers X9, X10, X19

2. Do Procedure

3. Restore registers
Jump to return
address

Leaf Procedure Example

```
leaf_example:
   SUBI SP,SP,#24
   STUR X10,[SP,#16]
   STUR X9,[SP,#8]
   STUR X19,[SP,#0]
```

Make space for 3 8-byte values on the stack

Save X10, X9, X19 on stack

1. Spill registers

2. Do Procedure

3. Restore registers
Jump to return
address

Leaf Procedure Example

1. Spill registers

2. Do Procedure

3. Restore registers
Jump to return
address

```
leaf_example:
  SUBI SP,SP,#24
  STUR X10,[SP,#16]
  STUR X9,[SP,#8]
  STUR X19,[SP,#0]
  ADD X9,X0,X1
  ADD X10,X2,X3
  SUB X19,X9,X10
  ADD X0,X19,XZR
```

Make space for 3 8-byte values on the stack

Save X10, X9, X19 on stack

X9 = g + h

X10 = i + j

f = X9 – X10

copy f to return register

Leaf Procedure Example

	`leaf_example:`	
	`SUBI SP,SP,#24`	Make space for 3 8-byte values on the stack
	`STUR X10,[SP,#16]`	
1. Spill registers	`STUR X9,[SP,#8]`	Save X10, X9, X19 on stack
	`STUR X19,[SP,#0]`	
	`ADD X9,X0,X1`	X9 = g + h
	`ADD X10,X2,X3`	X10 = i + j
2. Do Procedure	`SUB X19,X9,X10`	f = X9 – X10
	`ADD X0,X19,XZR`	copy f to return register
	`LDUR X10,[SP,#16]`	
	`LDUR X9,[SP,#8]`	Restore X10, X9, X19 from stack
3. Restore registers Jump to return address	`LDUR X19,[SP,#0]`	
	`ADDI SP,SP,#24`	Restore stack pointer
	`BR LR`	Return to caller

Registers to be Saved

- X9 to X17: temporary registers
  - The caller has to save them if needed for latter
  - Not preserved by the callee
- X19 to X28: saved registers
  - If used, the callee saves and restores them

What Registers are saved?

- X0 – X7: procedure arguments/results
- **X9 – X15: temporaries registers**
- **X19 – X27: saved registers**
- X28 (SP): stack pointer (address of the most recently allocated stack)
- X30 (LR): link register (return address)
  - Also called as program counter (PC)

Leaf Procedure Example

	`leaf_example:`	
	`SUBI SP,SP,#24`	Make space for 3 8-byte values on the stack
1. Spill registers	`STUR X10,[SP,#16]`	
	`STUR X9,[SP,#8]`	
	`STUR X19,[SP,#0]`	
	`ADD X9,X0,X1`	X9 = g + h
2. Do Procedure	`ADD X10,X2,X3`	X10 = i + j
	`SUB X19,X9,X10`	f = X9 – X10
	`ADD X0,X19,XZR`	copy f to return register
	`LDUR X10,[SP,#16]`	
	`LDUR X9,[SP,#8]`	Restore X10, X9, X19 from stack
3. Restore registers	`LDUR X19,[SP,#0]`	
Jump to return address	`ADDI SP,SP,#24`	Restore stack pointer
	`BR LR`	Return to caller

Leaf Procedure Example

```
leaf_example:
  SUBI SP,SP,#24
  STUR X10,[SP,#16]
  STUR X9,[SP,#8]
  STUR X19,[SP,#0]
  ADD X9,X0,X1
  ADD X10,X2,X3
  SUB X19,X9,X10
  ADD X0,X19,XZR
  LDUR X10,[SP,#16]
  LDUR X9,[SP,#8]
  LDUR X19,[SP,#0]
  ADDI SP,SP,#24
  BR LR
```

1. Spill registers

Make space for 3 8-byte values on the stack

Temporary registers, must be saved by the caller before calling the procedure

Saved by the callee

2. Do Procedure

X9 = g + h

X10 = i + j

f = X9 – X10

copy f to return register

3. Restore registers
Jump to return
address

Restore X10, X9, X19 from stack

Restore stack pointer

Return to caller

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call,
  - Caller pushed, argument (X0-X7) and save temporary register(X9-X17)
  - Callee pushes return address (LR) and saved registers (X19-X25)
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

  - Argument n in X0
  - Result in X1

Non-Leaf Procedure Example

- C code:

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

Callee saves LR and the argument (X0)

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
```

Save return address and n on stack

Non-Leaf Procedure Example

- C code:

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

Callee saves LR and the argument (X0)

  - Argument n in X0
  - Result in X1

Non-Leaf Procedure Example

- C code:

```c
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

Check inequality and branch to else (L1) or return 1

  - Argument n in X0
  - Result in X1

Non-Leaf Procedure Example

- C code:

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

Check inequality and branch to else (L1) or return 1

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16                       Save return address and n on stack
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1                      compare n and 1
   B.GE L1                              if n >= 1, go to L1
```

Non-Leaf Procedure Example

- C code:

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

Check inequality and branch to else (L1) or return 1

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1
   B.GE L1
   ADDI X1,XZR,#1
   ADDI SP,SP,#16
   BR LR
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don’t bother restoring values

Return

Non-Leaf Procedure Example

- C code:

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

Decrement n (X0) and call fact agian

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1
   B.GE L1
   ADDI X1,XZR,#1
   ADDI SP,SP,#16
   BR LR
L1: SUBI X0,X0,#1
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don’t bother restoring values

Return

n = n - 1

Non-Leaf Procedure Example

- C code:

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

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1
   B.GE L1
   ADDI X1,XZR,#1
   ADDI SP,SP,#16
   BR LR
L1: SUBI X0,X0,#1
   BL fact
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don’t bother restoring values

Return

n = n - 1

call fact(n-1)

Non-Leaf Procedure Example

- C code:

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

  - Argument n in X0
  - Result in X1

When fact returns, load the value of x0, and return address from stack

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1
   B.GE L1
   ADDI X1,XZR,#1
   ADDI SP,SP,#16
   BR LR
L1: SUBI X0,X0,#1
   BL fact
   LDUR X0,[SP,#0]
   LDUR LR,[SP,#8]
   ADDI SP,SP,#16
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don’t bother restoring values

Return

n = n - 1

call fact(n-1)

Restore caller’s n

Restore caller’s return address

Pop stack

Non-Leaf Procedure Example

- C code:

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

  - Argument n in X0
  - Result in X1

Leaf Procedure Example

```
fact:
   SUBI SP,SP,#16
   STUR LR,[SP,#8]
   STUR X0,[SP,#0]
   SUBIS XZR,X0,#1
   B.GE L1
   ADDI X1,XZR,#1
   ADDI SP,SP,#16
   BR LR
L1: SUBI X0,X0,#1
   BL fact
   LDUR X0,[SP,#0]
   LDUR LR,[SP,#8]
   ADDI SP,SP,#16
   MUL X1,X0,X1
   BR LR
```

Save return address and n on stack

compare n and 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don’t bother restoring values

Return

n = n - 1

call fact(n-1)

Restore caller’s n

Restore caller’s return address

Pop stack

return n * fact(n-1)

return

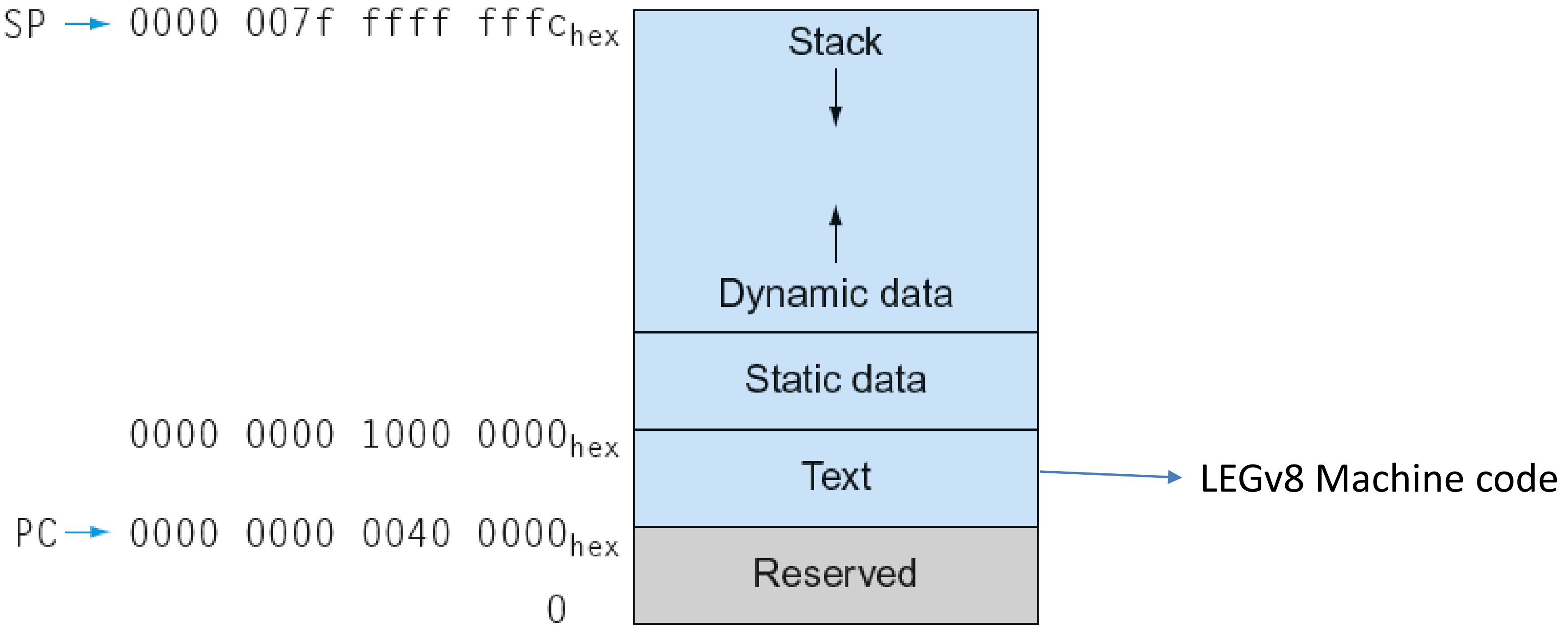
Frame Pointer

- Variables local to the procedure are also stored in the stack
- This segment of the stack is called procedure frame
- One of the 32 registers are used to store this address (Frame pointer)

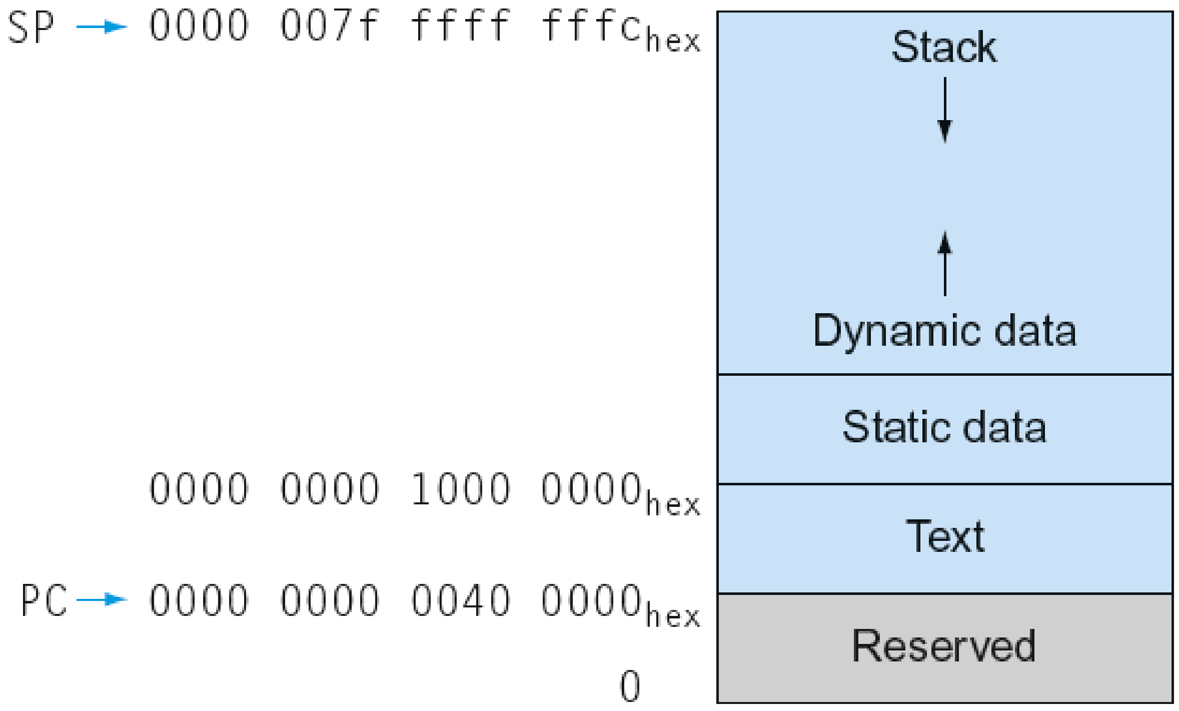
What Registers used?

- X0 – X7: procedure arguments/results
- X9 – X15: temporaries registers
- X19 – X27: saved registers
- X28 (SP): stack pointer (address of the most recently allocated stack)
- **X29 (FP): frame pointer**
- X30 (LR): link register (return address)
  - Also called as program counter (PC)

Memory Layout


SP → 0000 007f ffff fffc$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
LEGv8 Machine code
PC → 0000 0000 0040 0000$_{hex}$
Reserved
0

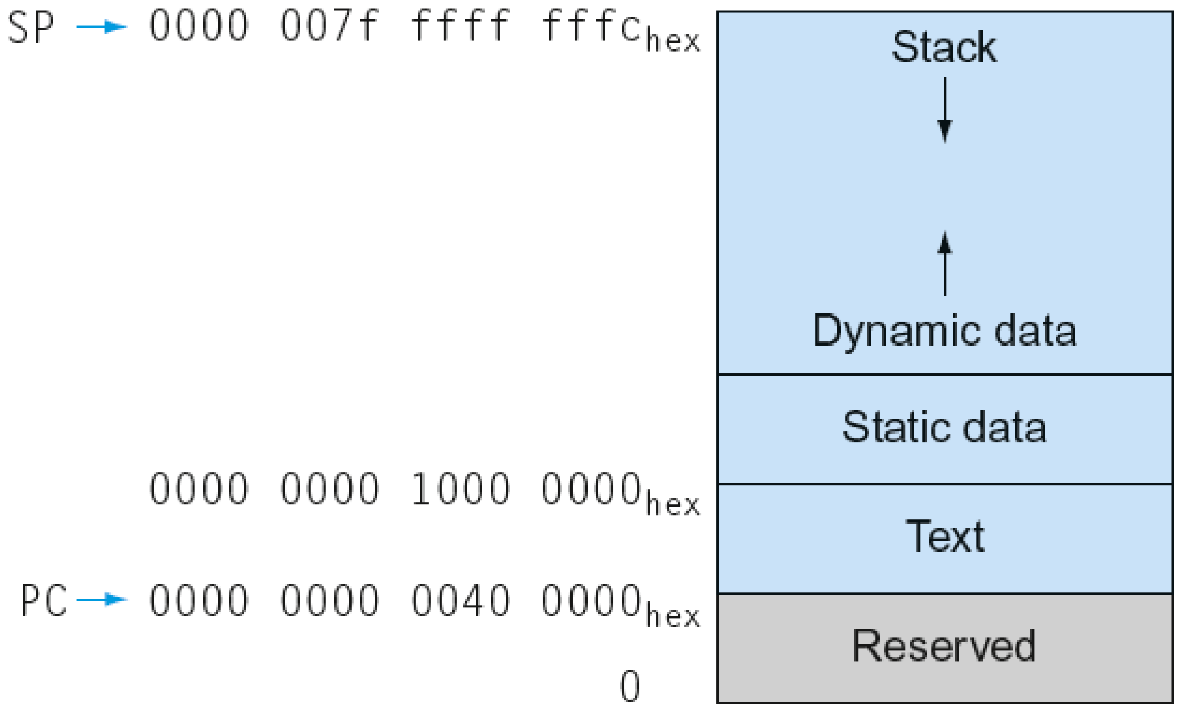
Memory Layout


SP → 0000 007f ffff fffc$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
PC → 0000 0000 0040 0000$_{hex}$
Reserved
0

→ Constants arrays in C, etc.

→ LEGv8 Machine code

Memory Layout

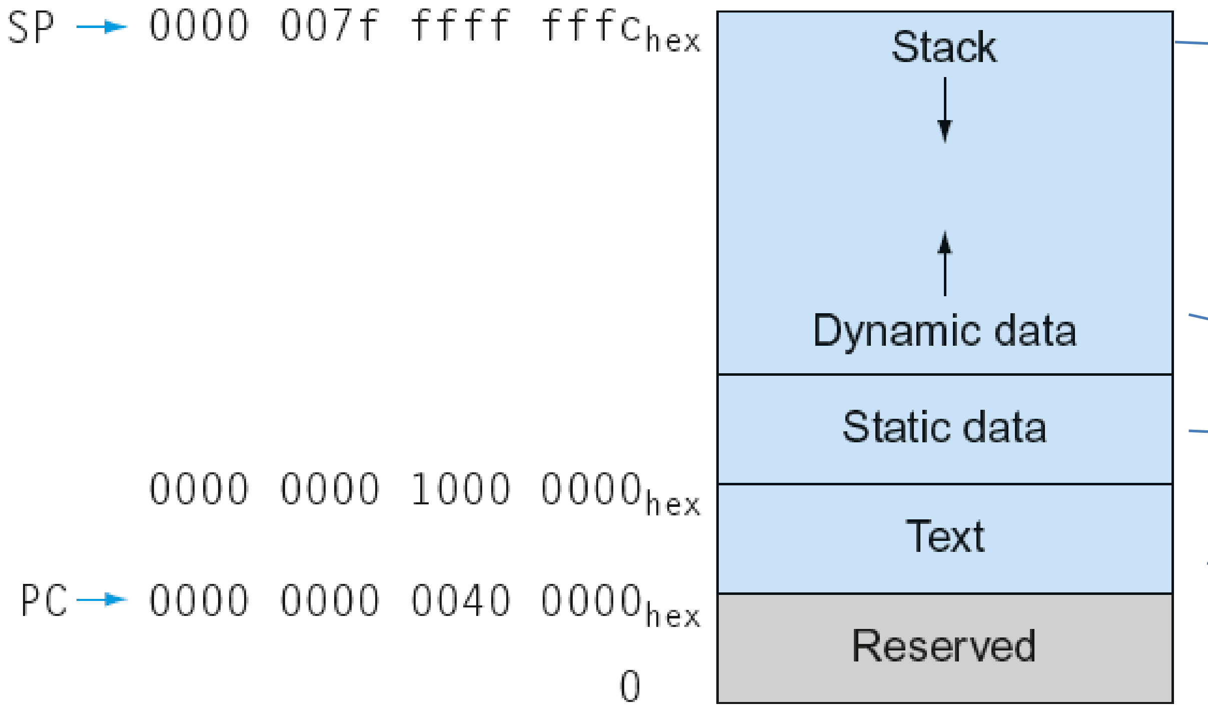

SP → 0000 007f ffff fffc$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
PC → 0000 0000 0040 0000$_{hex}$
Reserved
0

Dynamic data structures, linked list, etc. grows upwards

Constants arrays in C, etc.

LEGv8 Machine code

Memory Layout


SP
0000 007f ffff fffc$_{hex}$
Stack
Dynamic data
Static data
0000 0000 1000 0000$_{hex}$
Text
PC
0000 0000 0040 0000$_{hex}$
Reserved
0

Stack, grows downwards

Dynamic data structures, linked list, etc.
grows upwards

Constants arrays in C, etc.

LEGv8 Machine code

Communicating with People

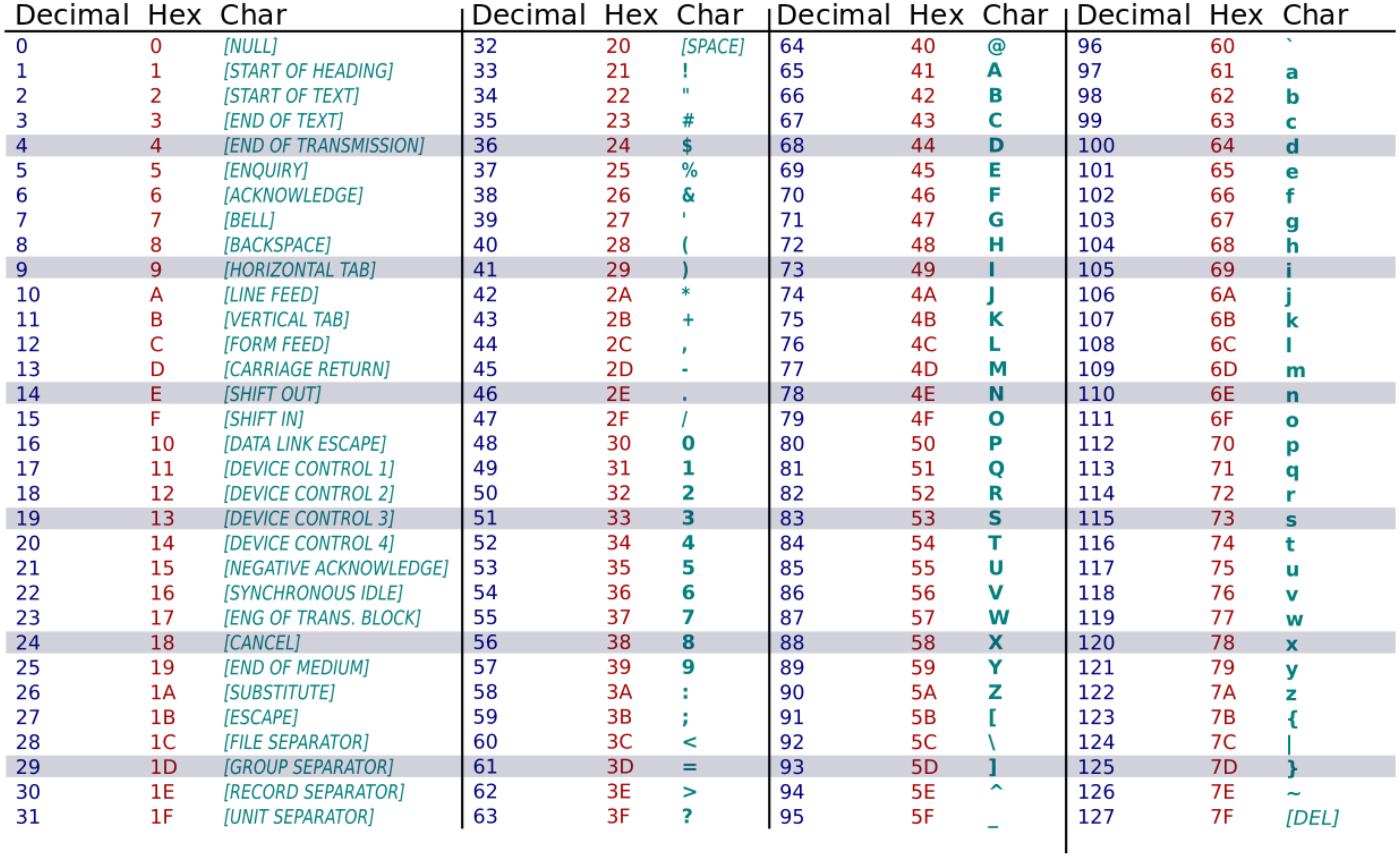
- Computers were initially invented to crunch numbers.
- Latter became commercially available are were used to process text.

- Use 8-bit bytes to represent character.

ASCII

- Acronym for the ***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange.
  - Seven-bit code proposed first by the American National Standards Institute (ANSI) in 1963, and finalized in 1968 as ANSI Standard X3.4.
  - The purpose of ASCII was to provide a standard to code various symbols (visible and invisible symbols)

ASCII TABLE



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Communicating with People

- Strings: Represented as series of characters (define start and end)
  - Reserve first position for length
  - An accompanying variable has length of string
  - Last position of the string has a special character
- C Programming uses ASCII and terminates string with 0
  - "Cal" ➔ 67,97,108,0

UNICODE

- Unicode is a universal encoding of the alphabets of human languages.

Unicode

- In ASCII a letter maps to a unique integer

  A -> 0100 0001

- In Unicode, a letter maps to something called a *code point* which is still just a theoretical concept.
- Example: simple string such as

  Hello

  corresponds in Unicode to these five *code points*

  U+0048 U+0065 U+006C U+006C U+006F

- It doesn’t say anything about how to store this in memory or represent it in an email message.

UNICODE Formats

- Unicode Transformation Format Encodings
  - UTF-2
  - UTF-7
  - UTF-8 (Most commonly used)
  - UTF-16 (Used by Java)
  - UTF-32

- C uses 8 bit bytes for characters
- Java uses 16 bits halfwords for characters
- LEGv8 provides instructions to load an store these formats
  - LDUR**B** (Load Byte – 8 bits)
  - STUR**B** (Store Byte – 8 bits)
  - LDUR**H** (Load Halfword – 16 bits)
  - STUR**H** (Store Halfword – 16 bits)

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR, **LDURB, STURB, LDURH, STURH**
Arithmetic Immediate	ADDI, SUBI, ORRI, ANDI, EORI
Logical Operations	LSL, LSR, AND, ORR, EOR
Branches	B, CBZ, CBNZ, B.Cond
Set Condition Flag	ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS
Procedure Instructions	BR, BL

LEGv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- Immediate instructions
  - Rn: source register
  - Rd: destination register

- Immediate field is zero-extended

What if we need a constant that is larger than 12 bits?

Wide Immediate Operands

- If a large constant is used
  - Compiler or assembler can assemble the value in a register and then use it.
  - This is used to specify the larger constants and addresses

Wide Immediate Operands

- LEGv8 instructions
  - MOVZ (Move wide and with zeros)
  - MOVK (Move wide with keep)

Wide Immediate Operands

- LEGv8 instructions
  - MOVZ (Move wide and with zeros)
  - MOVK (Move wide with keep)

Set 16 bits of constant in the register
MOVZ, zeros the rest of the bits
MOVK, keeps the rest of the bits

- Can specify to load any of the quadrant using in combination with LSL (0, 16, 32, 48)

Example

- $MOVZ\ X9\ 255\ LSL\ 0$

  1111 1111

$MOVZ\ X9\ 255\ LSL\ 0$

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111

$MOVZ\ X9\ 255\ LSL\ 16$

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000

$MOVZ\ X9\ 255\ LSL\ 32$

0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000	0000 0000 0000 0000

$MOVZ\ X9\ 255\ LSL\ 48$

0000 0000 1111 1111	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000

Example

- Add a constant 1902848 (64 bit representation)

00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000

What is the LEGv8 assembly code to load this 64-bit constant into register X19?

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

What is the LEGv8 assembly code to load this 64-bit constant into register `X19`?

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

First, we would load bits 16 to 31 with that bit pattern, which is 61 in decimal, using `MOVZ`:

What is the LEGv8 assembly code to load this 64-bit constant into register `X19`?

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

First, we would load bits 16 to 31 with that bit pattern, which is 61 in decimal, using `MOVZ`:

```
MOVZ     X19, 61, LSL 16 // 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `X19` afterward is:

```
00000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

What is the LEGv8 assembly code to load this 64-bit constant into register `X19`?

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

First, we would load bits 16 to 31 with that bit pattern, which is 61 in decimal, using `MOVZ`:

```
MOVZ    X19, 61, LSL 16 // 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `X19` afterward is:

```
00000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

What is the LEGv8 assembly code to load this 64-bit constant into register X19?

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

First, we would load bits 16 to 31 with that bit pattern, which is 61 in decimal, using MOVZ:

```
MOVZ     X19, 61, LSL 16 // 61 decimal = 0000 0000 0011 1101 binary
```

The value of register X19 afterward is:

```
00000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

The next step is to insert the lowest 16 bits, whose decimal value is 2304:

```
MOVK     X19, 2304, LSL 0 // 2304 decimal = 00001001 00000000
```

The final value in register X19 is the desired value:

```
00000000 00000000 00000000 00000000 00000000 00111101 00001001 00000000
```

- The representation for 1902848 is in X19.

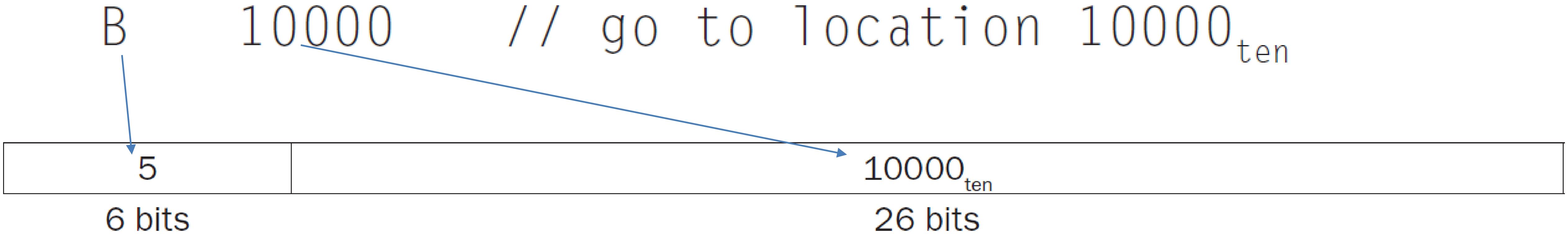
Branch Formats

- Unconditional Branches

```
B    10000     // go to location 10000ten
```

Branch Formats

- Unconditional Branches



```
B     10000     // go to location 10000ten
```

5	10000_{ten}
6 bits	26 bits

B-Type

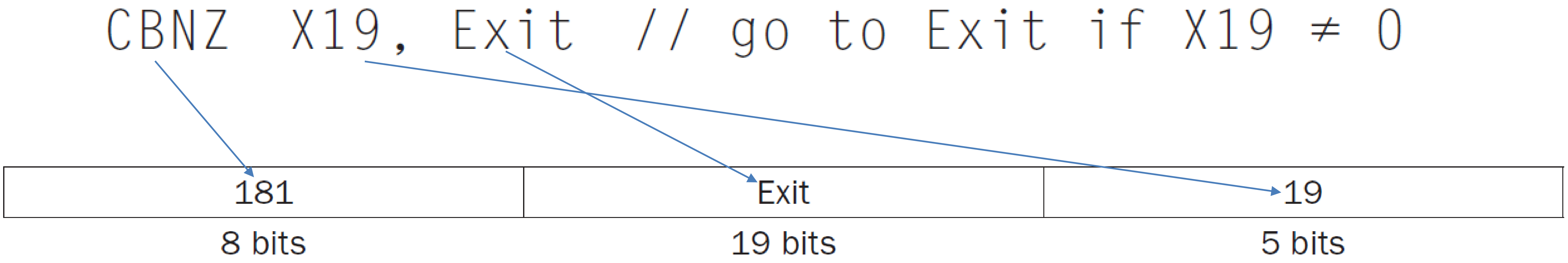
Branch Formats

- Conditional Branches

```
CBNZ  X19, Exit  // go to Exit if X19 ≠ 0
```

Branch Formats

- Conditional Branches

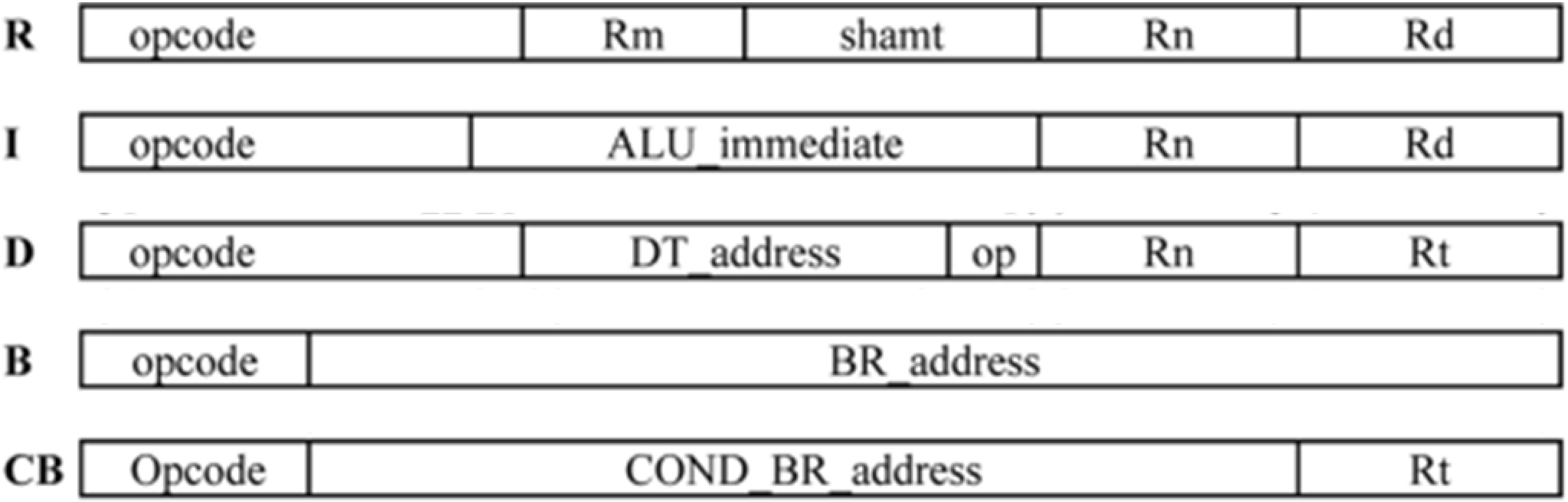


```
CBNZ  X19, Exit  // go to Exit if X19 ≠ 0
```

181	Exit	19
8 bits	19 bits	5 bits

CB-Type

Formats



R	opcode	Rm	shamt	Rn	Rd
I	opcode	ALU_immediate	Rn	Rd	
D	opcode	DT_address	op	Rn	Rt
B	opcode	BR_address			
CB	Opcode	COND_BR_address	Rt		

Addressing in Branches

181	Exit	19
8 bits	19 bits	5 bits

- Used to specify address
- Program cannot be larger than 2^{19}
  - – Too small, not realistic

Addressing in Branches

181	Exit	19
8 bits	19 bits	5 bits

- Used to specify address
- Program cannot be larger than 2^{19}
  - Too small, not realistic
- **Alternatively** store relative address in a register (64 bits) and add to a base address
  - Program can be as large as 2^{64}

Addressing in Branches

181	Exit	19
8 bits	19 bits	5 bits

- Used to specify address
- Program cannot be larger than 2^{19}
  - Too small, not realistic
- **Alternatively** store relative address in a register (64 bits) and add to a base address
  - Program can be as large as 2^{64}
  - PC, program counter is used as the base address

PC Relative addressing

$$Branch\ Address = Program\ Counter + register\ (offset)$$

$$\left.\begin{array}{l}\text{B- Type}\\[1em]\text{CB-Type}\end{array}\right\}\ \text{Both use pc relative addressing}$$

Addressing Modes

$$ADDI\ X19, X19, \#10$$

Addressing Modes

1. **Immediate addressing**: operand is a constant

$$ADDI\ X19, X19, \underline{\#10}$$

Addressing Modes

1. Immediate addressing: operand is a constant
2. **Register addressing**: operand is a register

$$ADDI\ X19, \underline{X19}, \#10$$

Addressing Modes

1. Immediate addressing: operand is a constant
2. **Register addressing**: operand is a register

$$LDUR\ X19, \underline{[X10, \#16]}$$

Addressing Modes

1. Immediate addressing: operand is a constant
2. Register addressing: operand is a register
3. **Base or Displacement addressing:** memory location (sum of register (X10) and constant (10))

$$LDUR\ X19, \underline{[X10, \#16]}$$

Addressing Modes

1. Immediate addressing: operand is a constant
2. Register addressing: operand is a register
3. Base or Displacement addressing: memory location (sum of register (X10) and constant (10))

$$CB\ \underline{1000_{ten}}$$

Addressing Modes

1. Immediate addressing: operand is a constant
2. Register addressing: operand is a register
3. Base or Displacement addressing: memory location (sum of register (X10) and constant (10))
4. **PC relative addressing:** Sum of PC and constant

$$CB\ \underline{1000_{ten}}$$