Computer Organization and Architecture

COSC 2425

Lecture – 8

Sept 14th , 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

Chapter 2

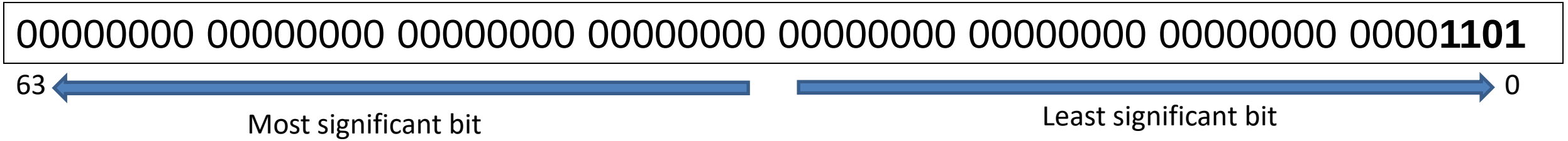
Instructions: Language of the Computer

Review

LEGv8: Signed and Unsigned Numbers

- LEGv8: 64 bit double word representation.

- Example Representation: $11_{ten} = 1101_{two}$



Review

LEGv8: Unsigned Numbers

- LEGv8: 64 bit double word representation.
  - Can represent 2^{64} different patterns.
- Numbers range from $[0, 2^{64} - 1](18{,}446{,}774{,}073{,}709{,}551{,}615_{ten})$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two} = 0_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}$

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{two} = 2_{ten}$

. . . . . .

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{two} = 18{,}446{,}774{,}073{,}709{,}551{,}613_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}614_{ten}$

$11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} = 18{,}446{,}744{,}073{,}709{,}551{,}615_{ten}$

$$(x_{63} \times 2^{63}) + (x_{62} \times 2^{62}) + (x_{61} \times 2^{61}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Review

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2's complement** representation.

Review

LEGv8: Signed Numbers

- LEGv8: 64 bit double word **2's complement** representation.
- Positive half range:
  - $[0\ to\ 9{,}223{,}372{,}036{,}854{,}775{,}807_{ten}]$
- Negative half
  - $[-1\ to\ -9{,}223{,}372{,}036{,}854{,}775{,}808_{ten}]$

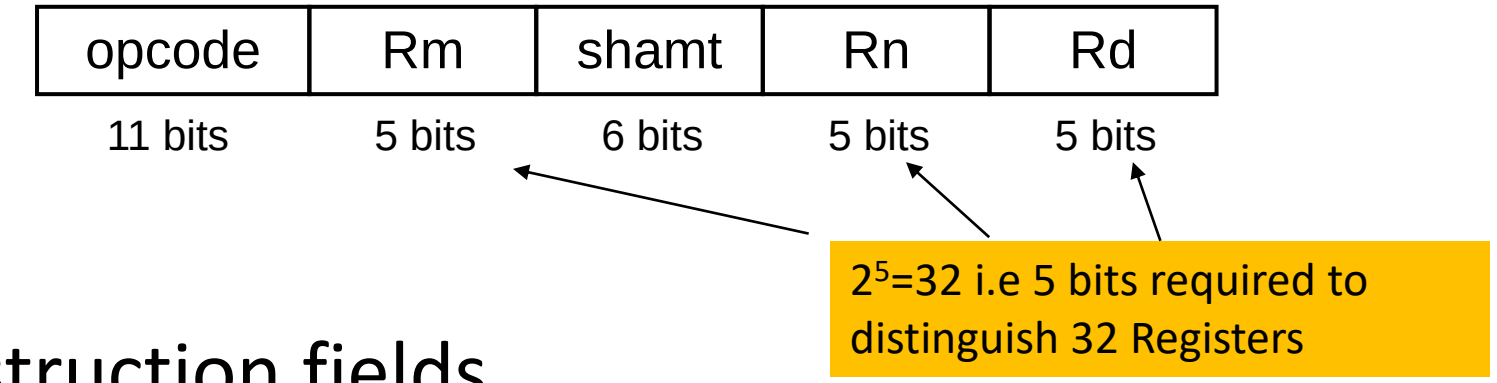
Review

Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- **LEGv8** instructions
  - Encoded as **32-bit instruction words**
  - **Different formats exists (but a small number)**
    - **R-Type ➔Arithmetic**
    - **D-Type ➔ Data transfer**
    - **I-Type ➔ Immediate**
    - **…**
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

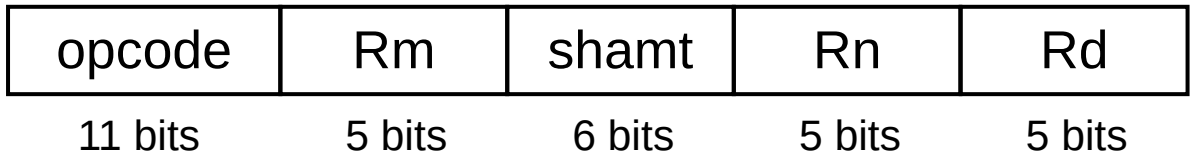
LEGv8 R-format Instructions



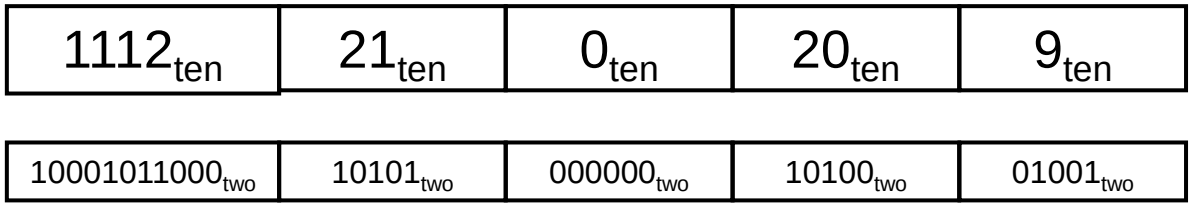
- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

Review

R-format Example



ADD X9,X20,X21



$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$ =

$8B150289_{16}$

Review

Different format for Data Transfer (D-Type)

- ***Design Principle 3****:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Review

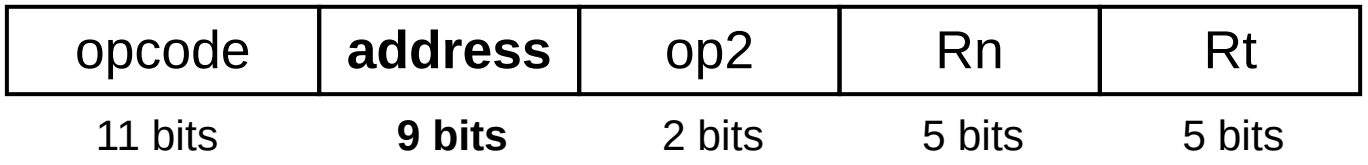
LEGv8 D-format Instructions

opcode	**address**	op2	Rn	Rt
11 bits	**9 bits**	2 bits	5 bits	5 bits

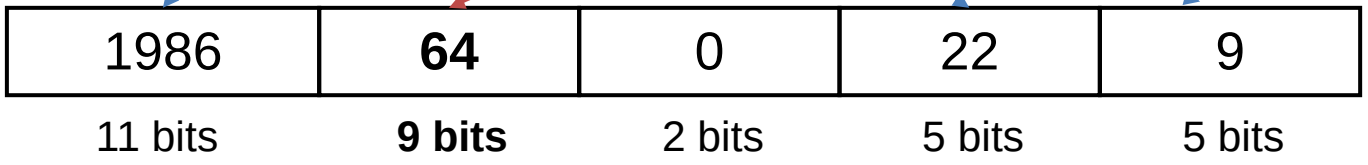
- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number

Review

Example: D-Type

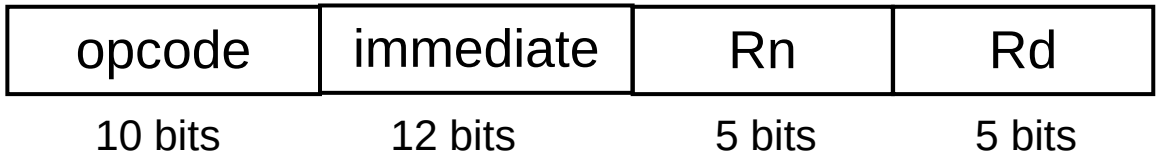


$$LDUR\ X9, [X22, \#\mathbf{64}]$$



Review

LEGv8 I-format Instructions



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

- Immediate field is zero-extended

Review

Example

$$A[30] = h + A[30] + 1$$

Base address of A stored in X10, h stored in X21

LEGv8 Assembly code:

LDUR X9, [X10, #240]
ADD X9, X21, X9
ADDI X9 ,X9, #1
STUR X9, [X10, #240]

Machine Language in Decimal:

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

Review

Example

$$A[30] = h + A[30] + 1$$

opcode	Rm/address	shamt/op2	Rn	Rd/Rt
1986	240	0	10	9
1112	9	0	21	9
580	1		9	9
1984	240	0	10	9

11111000010	011110000	00	01010	01001
10001011000	01001	000000	10101	01001
1001000100	000000000001		01001	01001
11111000000	011110000	00	01010	01001

Review

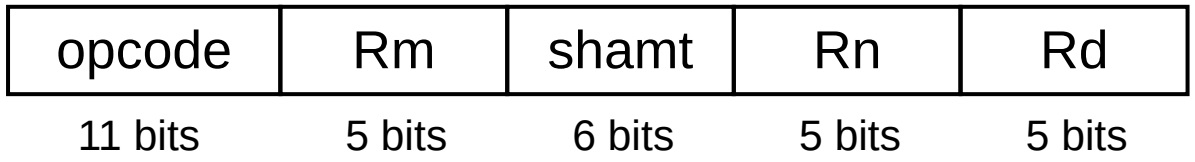
Logical Operations

- Instructions for bitwise manipulation

	Operation	C	Java	LEGv8
Shifts	Shift left	<<	<<	LSL
	Shift right	>>	>>	LSR
	Bit-by-bit AND	&	&	AND, ANDI
	Bit-by-bit OR	|	|	OR, ORI
	Bit-by-bit NOT	~	~	EOR, EORI

- Operate on bits/bytes more useful than on words
  - Examine characters (8 bits) within a word
- Useful for extracting and inserting groups of bits in a word

Shift Operations



- Use R- format
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - LSL Logical shift left
- Shift right logical
  - Shift right and fill with 0 bits
  - LSR Logical shift right

Review

Example LSL

$$LSL\ X11, X19, \#4 // shift\ 4\ bits\ to\ left$$

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$$

$$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 10010000_{two} = 144_{ten}$$

$$144_{ten} = 9_{ten} * 2^4$$

Left Shift by *i* bits multiplies by 2^i

Review

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

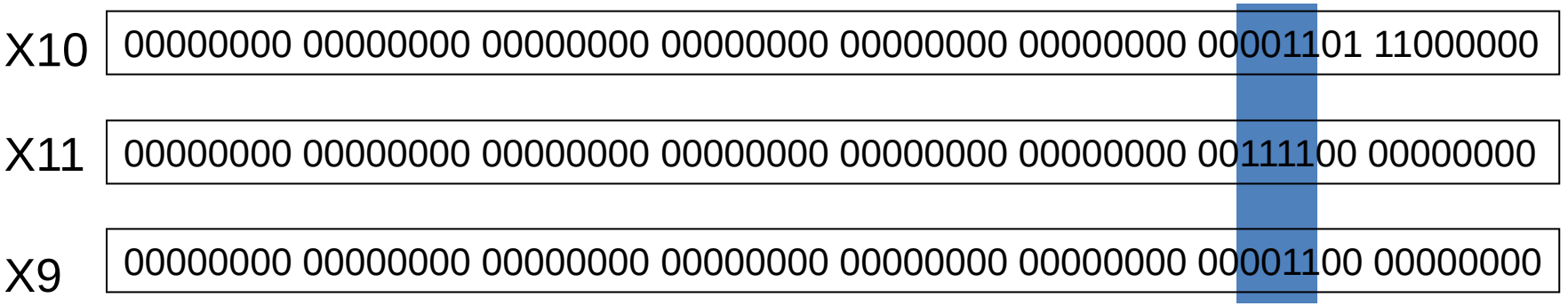
- Operate on bits/bytes more useful than on words
  - Examine characters (8 bits) within a word
- Useful for extracting and inserting groups of bits in a word

Review

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
AND  X9,X10,X11
```

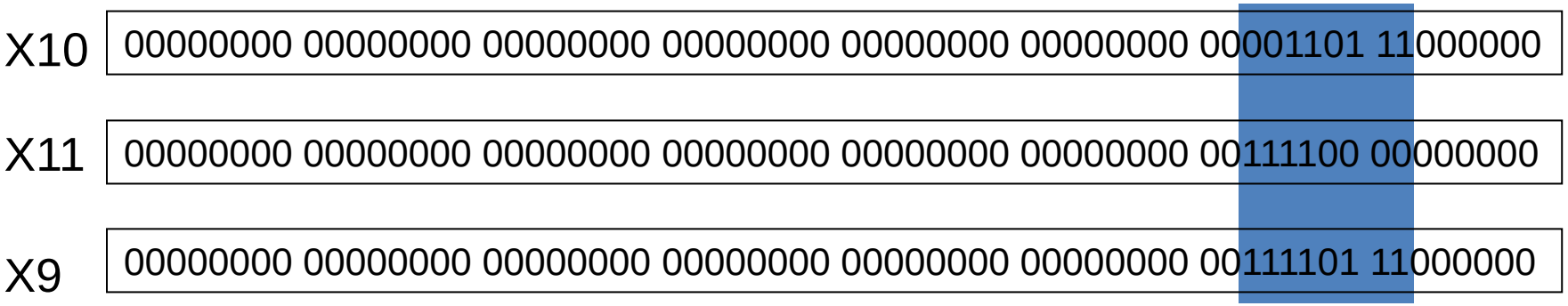


Review

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
ORR X9,X10,X11
```

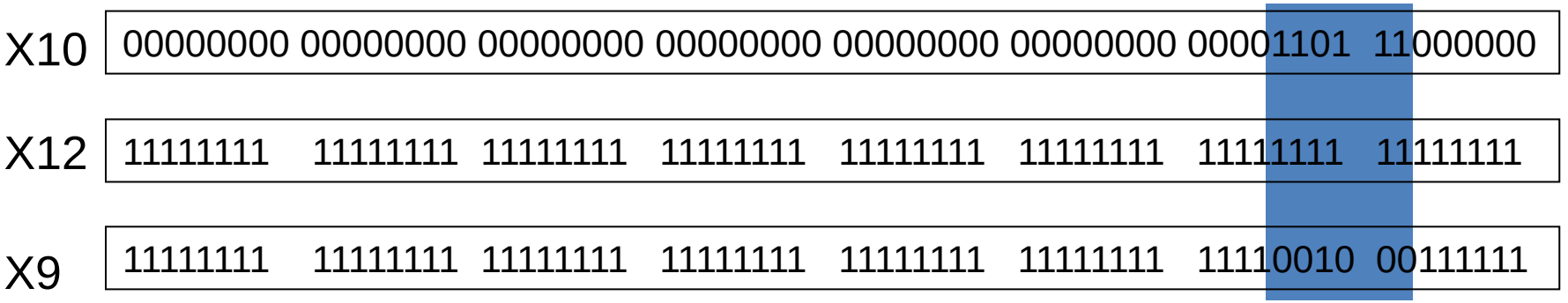


Review

EOR Operations

- Exclusive OR instead of NOT
- Differencing operation
  - Set some bits to 1, leave others unchanged

```
EOR X9,X10,X12  // NOT operation
```



- LEGv8 Code: ?

C code:

```
if ( i==j )
        f = g+h;
else
        f = g-h;
```

Review

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

  Label

- Labels are only for Assembly language
- Assembler changes them to address in machine code

Review

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Unconditional Branch: Instruct computer to branch to label
- B – branch to label
- LEGv8 Code:

  B L1 `// Branch to statement with label L1`

Review

Instructions for Making Decisions

- Define Labels for instructions.
- LEGv8 Code:

  L1: *ADD X9, X21, X9*

- Instruct computer to branch to instruction using the label if some condition is satisfied.
- CBZ – compare and branch if zero
- CBNZ – compare and branch if not zero
- LEGv8 Code:

  $CBZ\ \ register,\ \ L1$ // if (register == 0) branch to instruction labeled L1;

  $CBNZ\ \ register,\ \ L1$ // if (register != 0) branch to instruction labeled L1;

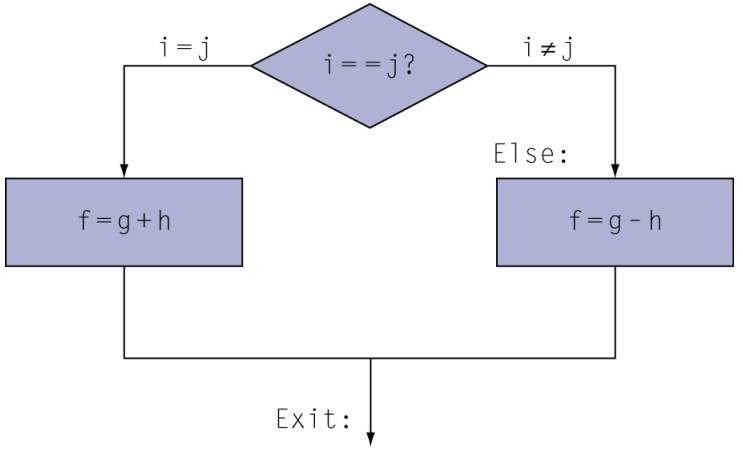
Review

Example: Compiling If Statements

- C code:

```
if ( i==j )
    f = g+h;
else
    f = g-h;
```

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21



- Compiled LEGv8 code:

X9 will be zero if i=j

Review

Example: Compiling If Statements

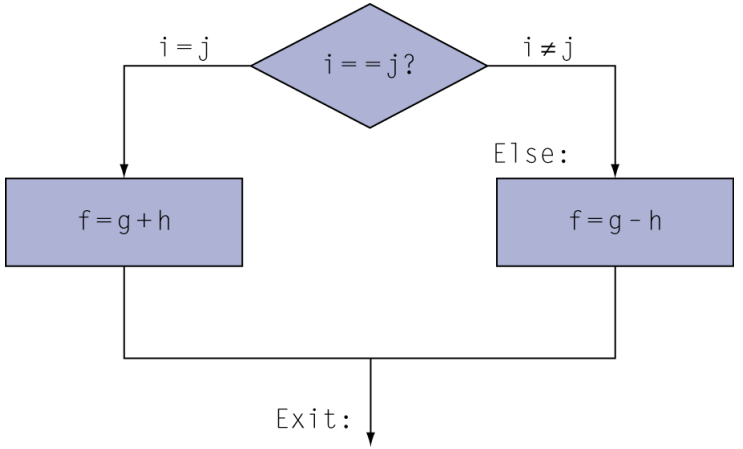
- C code:

```
if ( i==j )
    f = g+h;
else
    f = g-h;
```

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21

- Compiled LEGv8 code:

```
            SUB X9,X22,X23
            CBNZ X9,Else
            ADD X19,X20,X21
            B Exit
Else:       SUB X19,X20,x21
Exit: …
```



X9 will be zero if i=j

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
Arithmetic Immediate	ADDI, SUBI
Logical Operations	**LSL, LSR, AND, ORR, EOR**
Branches	**B, CBZ, CBNZ**

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
Arithmetic Immediate	ADDI, SUBI, **ANDI, ORRI, EORI**
Logical Operations	**LSL, LSR, AND, ORR, EOR**
Branches	**B, CBZ, CBNZ**

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
?
```

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:


        B       Loop                // uncond. branch
```

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:


        ADDI   X22,X22,#1        // i += 1
        B      Loop             // uncond. branch
```

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:
        ADDI   X22,X22,#1          // i += 1
        B      Loop               // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	Address
0	0	?

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:
        ADDI   X22,X22,#1          // i += 1
        B      Loop               // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	Address
0	0	X25
1	1	X25+8

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:
        ADDI   X22,X22,#1          // i += 1
        B      Loop               // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	Address
0	0	X25
1	1	X25+8
2	2	X25+2*8
3	3	X25+3*8

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop:
        ADDI   X22,X22,#1          // i += 1
        B      Loop               // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	X10	Address
0	0	0*8	X25+X10
1	1	1*8	X25+X10
2	2	2*8	X25+X10
3	3	3*8	X25+X10

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL   X10,X22,#3          // X10 = i*2^3

      ADDI  X22,X22,#1          // i += 1
      B     Loop              // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	X10	Address
0	0	0*8	X25+X10
1	1	1*8	X25+X10
2	2	2*8	X25+X10
3	3	3*8	X25+X10

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL   X10,X22,#3        // X10 = i*2^3
      ADD   X10,X10,X25       // Address to load save[i]

      ADDI  X22,X22,#1        // i += 1
      B     Loop             // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	X10	Address
0	0	0*8	X25+X10
1	1	1*8	X25+X10
2	2	2*8	X25+X10
3	3	3*8	X25+X10

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL   X10,X22,#3        // X10 = i*2^3
      ADD   X10,X10,X25       // Address to load save[i]
      LDUR  X9,[X10,#0]       // load save[i]

      ADDI  X22,X22,#1        // i += 1
      B     Loop             // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	X10	Address
0	0	0*8	X25+X10
1	1	1*8	X25+X10
2	2	2*8	X25+X10
3	3	3*8	X25+X10

Compiling Loop Statements

- C code:

```
while (True)
        k = k + save[i]
        i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL   X10,X22,#3        // X10 = i*2^3
      ADD   X10,X10,X25       // Address to load save[i]
      LDUR  X9,[X10,#0]       // load save[i]
      ADD   X24, X24, X9
      ADDI  X22,X22,#1        // i += 1
      B     Loop             // uncond. branch
```

Address to access Save[i]

Loop	I (X22)	X10	Address
0	0	0*8	X25+X10
1	1	1*8	X25+X10
2	2	2*8	X25+X10
3	3	3*8	X25+X10

Compiling Loop Statements

- C code:

```
while (save[i] == k)
    i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
?
```

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL    X10,X22,#3      // X10 = i*2^3
      ADD    X10,X10,X25     // Address to load save[i]
      LDUR   X9,[X10,#0]     // load save[i]


      ADDI   X22,X22,#1      // i += 1
      B      Loop            // uncond. branch
```

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

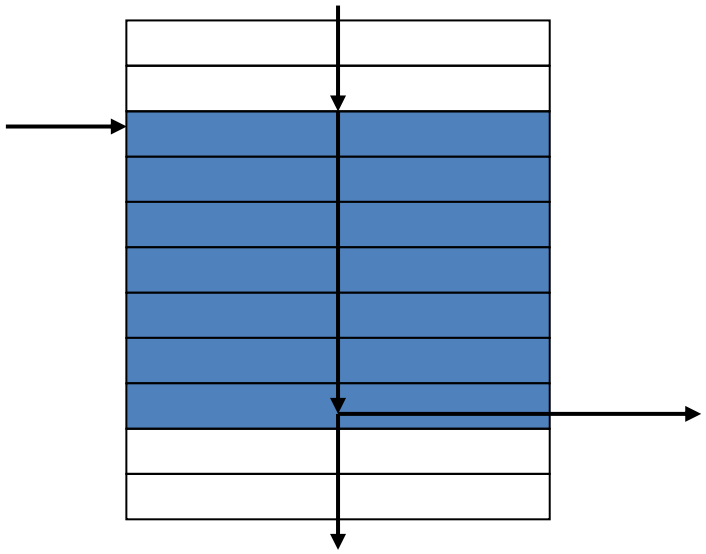
  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL    X10,X22,#3      // X10 = i*2^3
      ADD    X10,X10,X25     // Address to load save[i]
      LDUR   X9,[X10,#0]     // load save[i]
      SUB    X11,X9,X24      // X11 = save[i] - k
      CBNZ   X11,Exit        // conditional branch
      ADDI   X22,X22,#1      // i += 1
      B      Loop            // uncond. branch
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Condition Codes or Flags

- C code:

```
if ( i==j )
    f = g+h;
else
    f = g-h;
```

  - i, j in X22, X23,
  - f, g, h, in X19, X20, X21
- Compiled LEGv8 code:

```
            SUB X9,X22,X23
            CBNZ X9,Else
            ADD X19,X20,X21
            B Exit
Else:       SUB X19,X20,x21
Exit: …
```

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

```
Loop: LSL    X10,X22,#3
      ADD    X10,X10,X25
      LDUR   X9,[X10,#0]
      SUB    X11,X9,X24
      CBNZ   X11,Exit
      ADDI   X22,X22,#1
      B      Loop
Exit: …
```

Other Comparison

$<$	Less than
$\leq$	Less than or equal
$>$	Greater than
$\geq$	Greater than or equal
$=$	Equal
$!=$	Not equal

Other Comparison

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

```
if ( i==j )
..

SUB X9,i,j    // Subtract
CBNZ X9,Else // Check if zero or not
```

Other Comparison

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

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
```

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
How to check if X9 is negative?
```

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
How to check if X9 is negative?

Depends on if i,j are signed or
unsigned.
```

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
How to check if X9 is negative?

Depends on if i,j are signed or
unsigned.

If signed, check the first bit (if it
is 1 it is negative, else positive)
```

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
How to check if X9 is negative?

Depends on if i,j are signed or
unsigned.

If signed, check the first bit (if it
is 1 it is negative, else positive).

If unsigned, numbers are borrowed all
the way to the significant bit.
```

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.
```

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

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

$$If\ (i < j)$$

…

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:

$$SUBS\ X1, X1, X2$$

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:

$$SUBS\ X1, X1, X2$$

Result is -1
The N flag is set

Condition codes/flags

$Negative$(N)	1
$Zero$ (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 Branches that use Flags

- Format ➔ B.cond
- Use subtract to set flags and then conditionally branch

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Conditional Example

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

– a in X22, b in X23

LEGv8 Code:

?

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:
```

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
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**

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;
}
```

C Example

```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int ret;
7
8     ret = add(a, b);
9
10    return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16    int result;
17
18    result = num1 + num2
19
20    return result;
21 }
```

Has parameters

Has return value

C Example

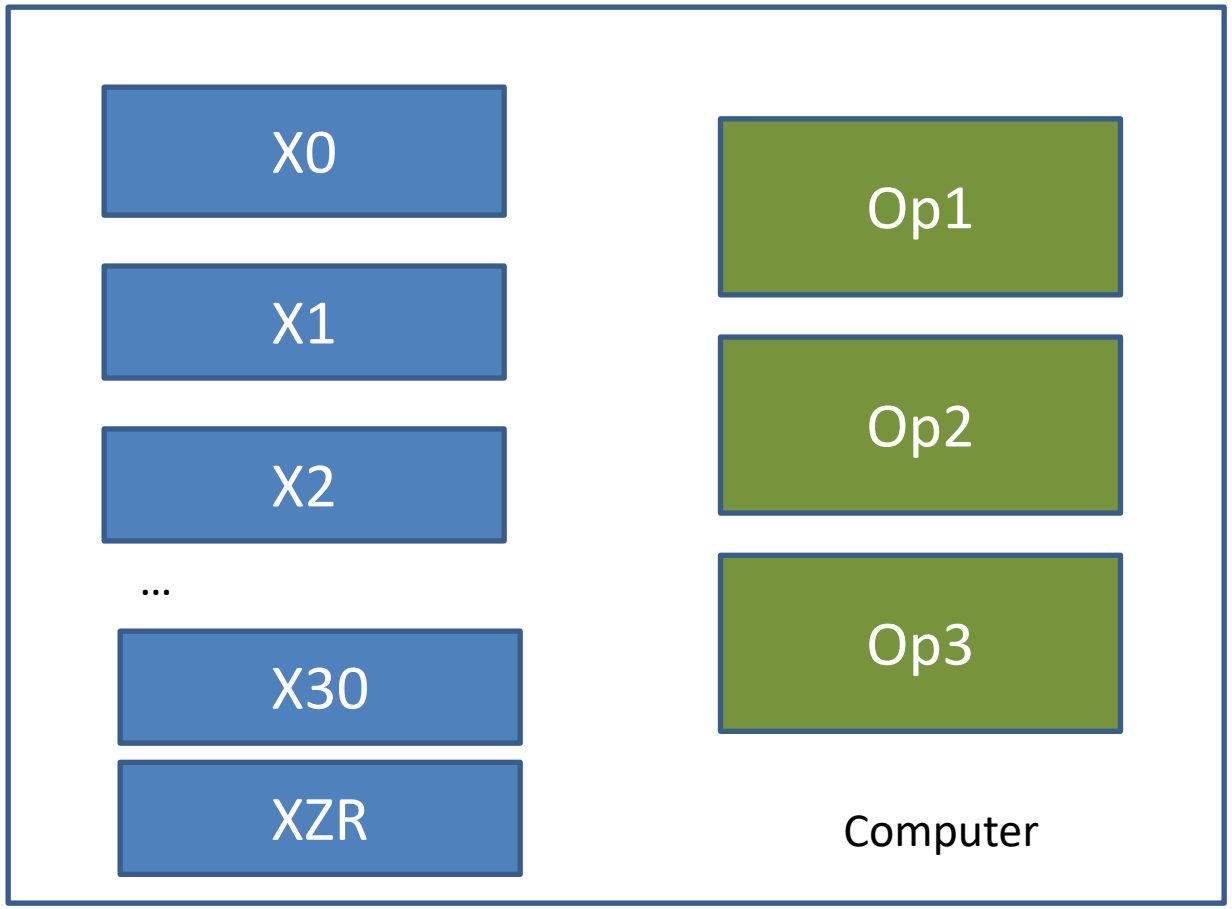
```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int ret;
7
8     ret = add(a, b);
9
10    return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16    int result;
17
18    result = num1 + num2
19
20    return result;
21 }
```

Caller

Callee

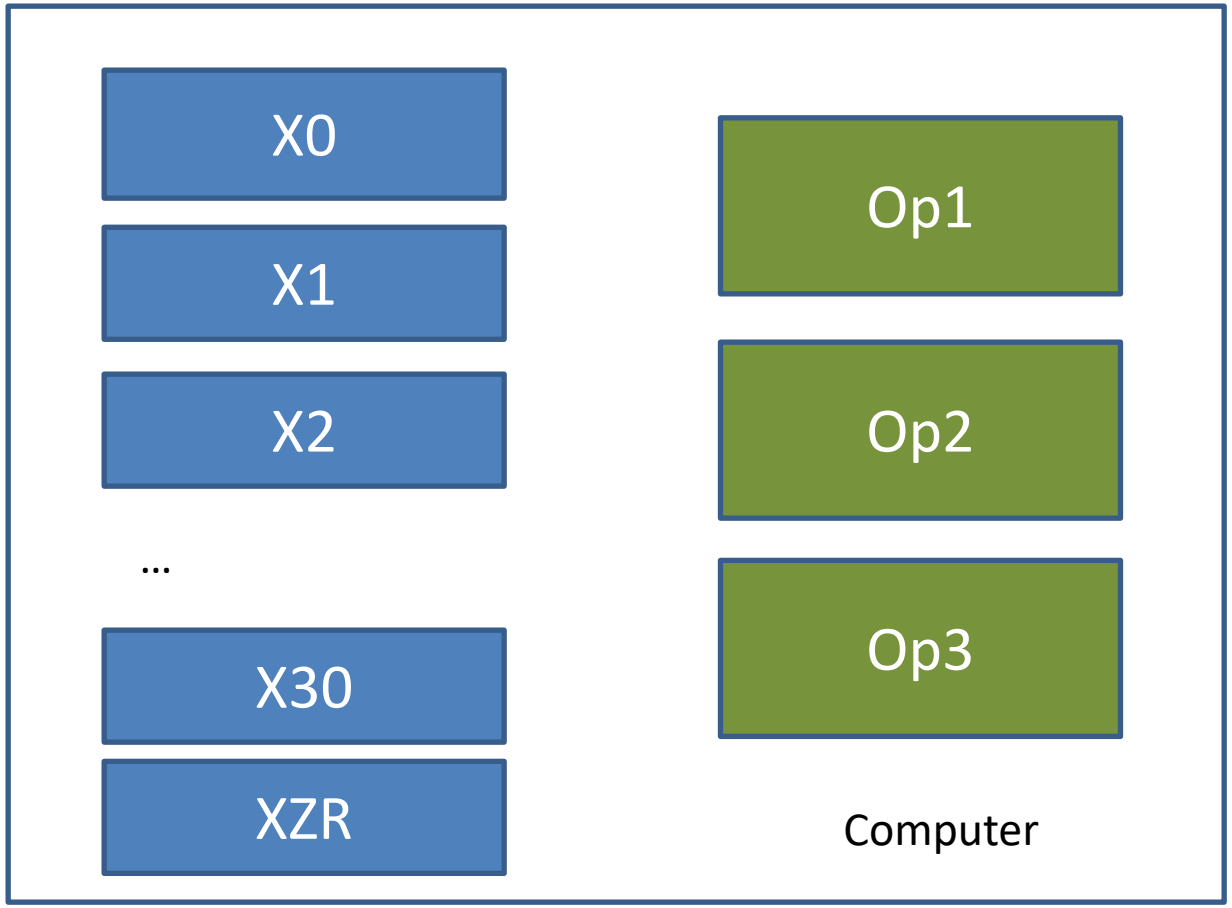
Steps in Executing a Procedure

```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



Steps in Executing a Procedure

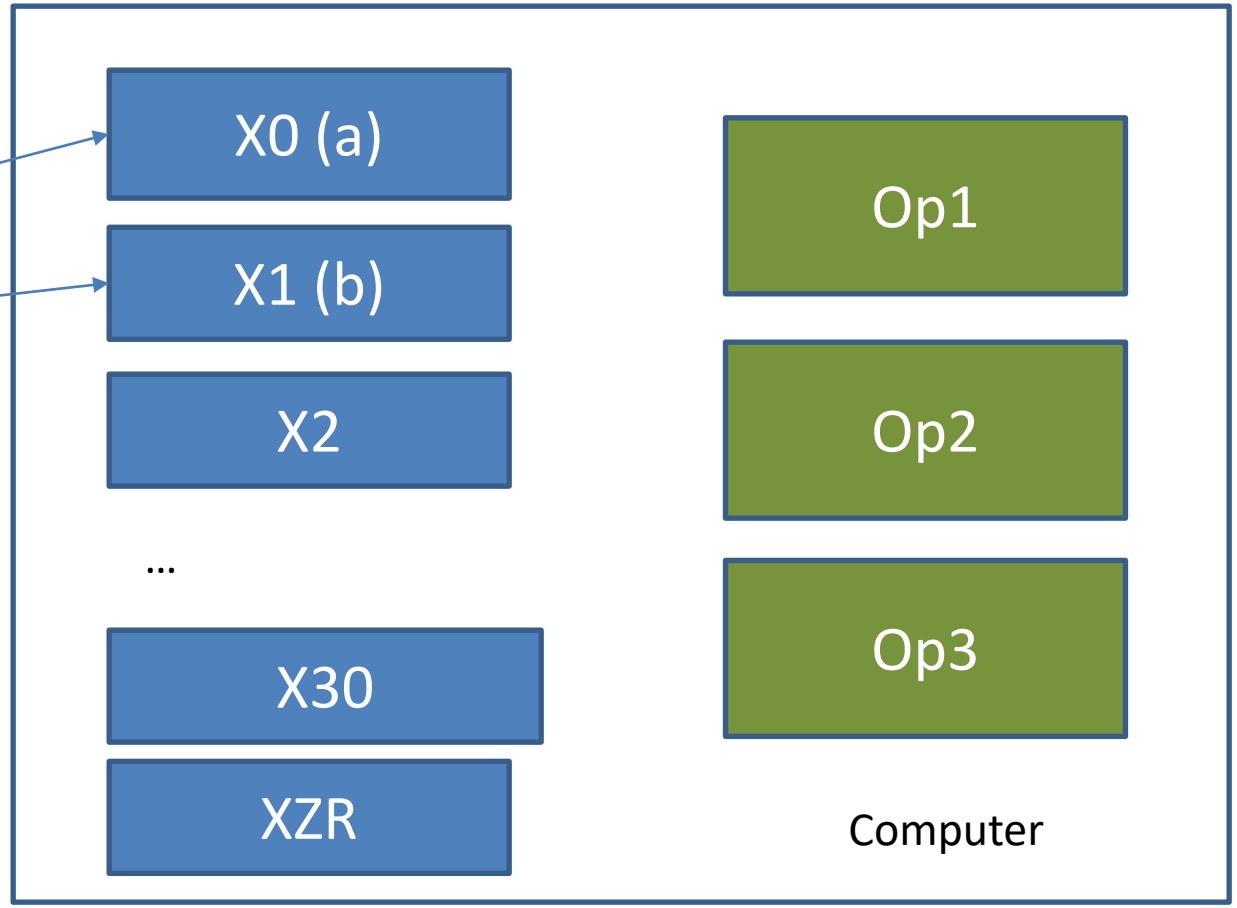
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



Initially main (Caller) has control of the computer.

Steps in Executing a Procedure

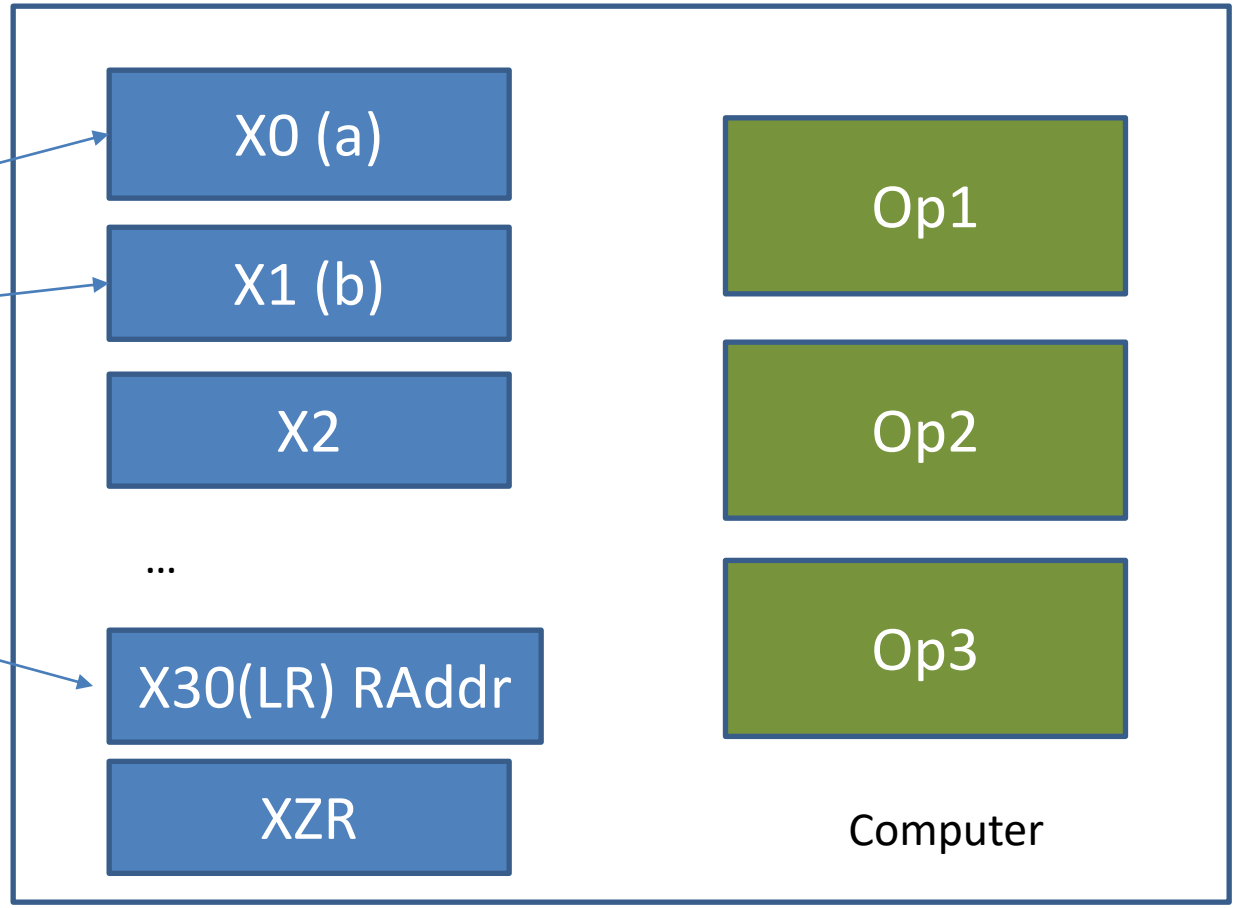
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ---------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



1. Caller puts parameters in registers for the procedure

Steps in Executing a Procedure

```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



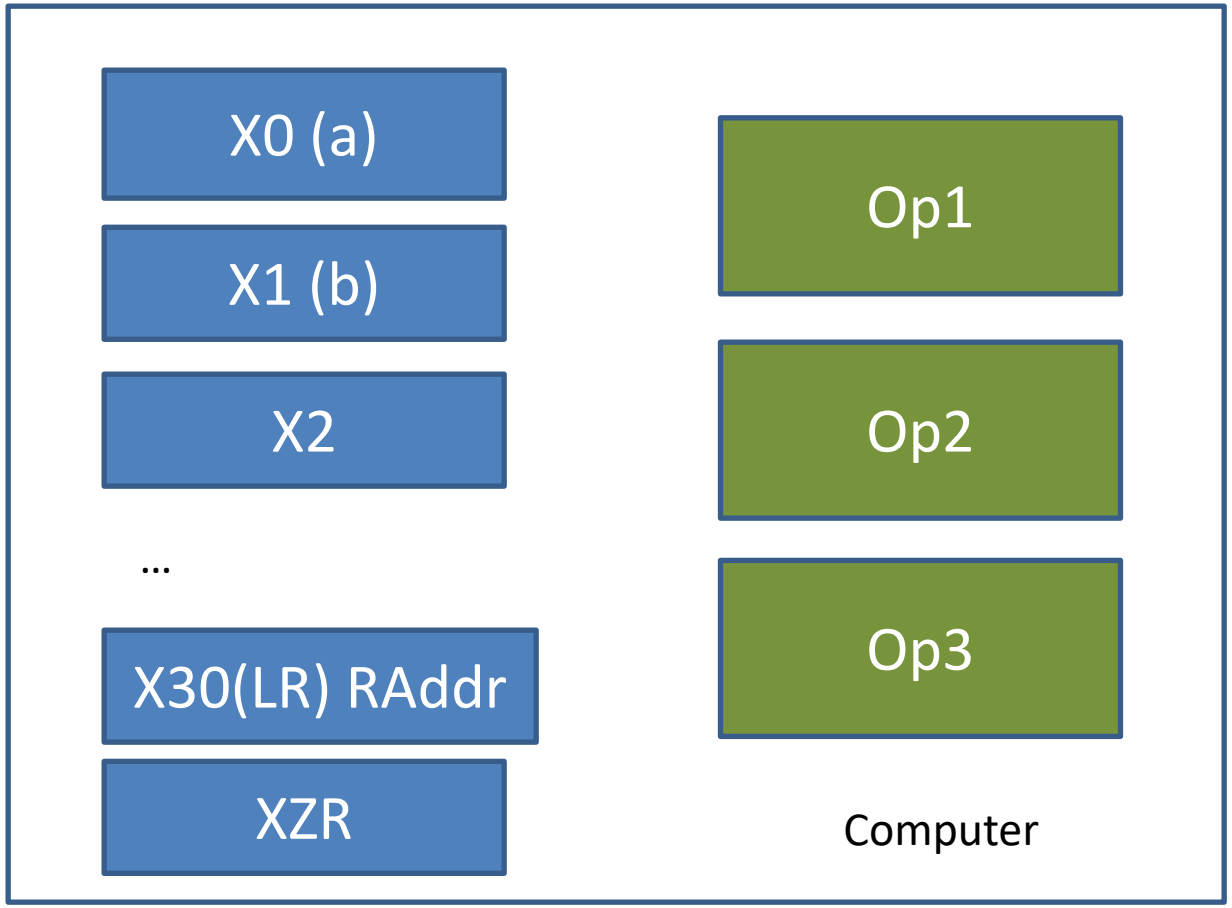
1. Caller puts parameters in registers for the procedure
    1. Specify a return address

What Registers used?

- X0 – X7: procedure **arguments/results**
- X30 (LR): link register (return address)
  - Also called as program counter (PC)
  - Program Counter: Contains the address of the current instruction

Steps in Executing a Procedure

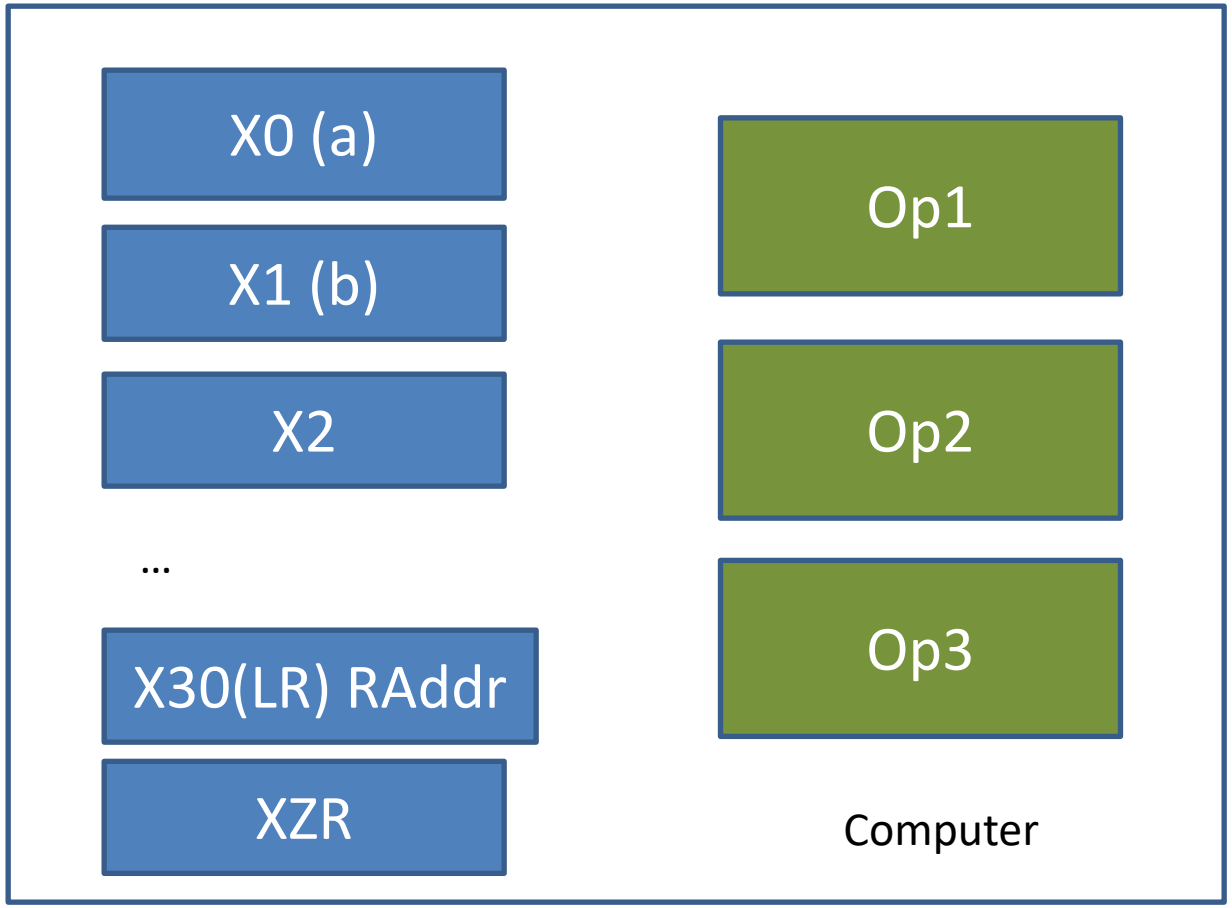
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



2. Transfer control to callee

Steps in Executing a Procedure

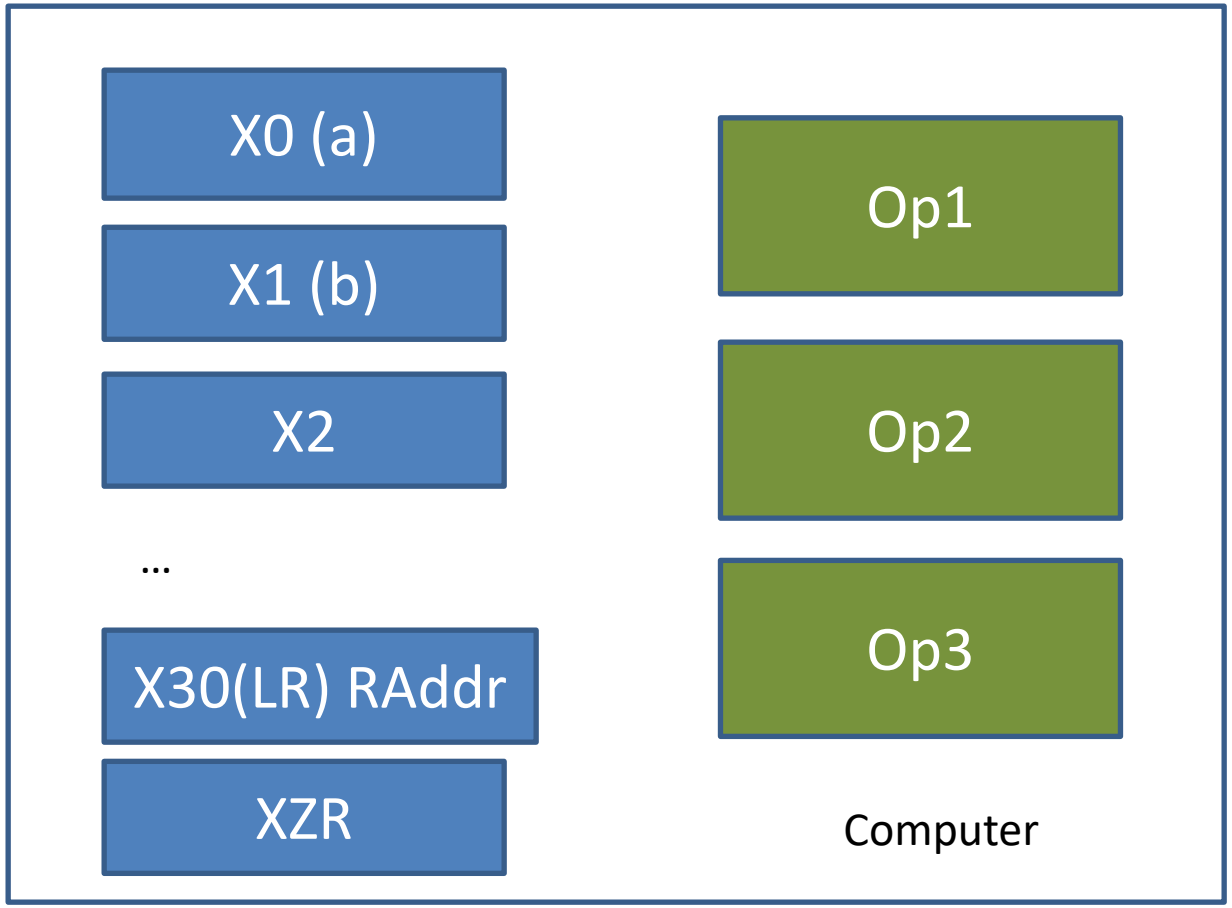
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ---------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



3. Acquire storage resources need for procedure

Steps in Executing a Procedure

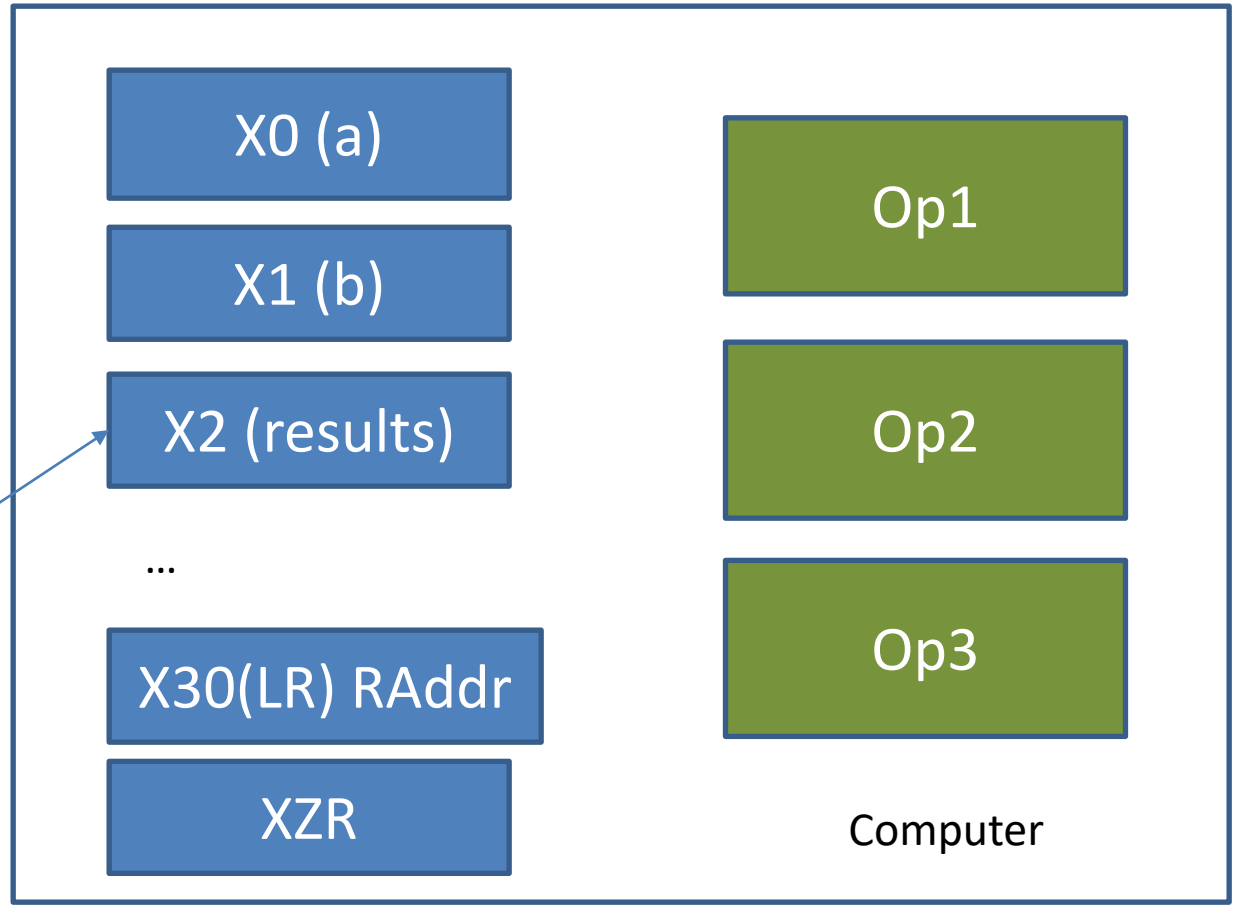
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ---------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



4. Perform Task

Steps in Executing a Procedure

```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int ret;
7  ---------------------------------
8     ret = add(a, b);
9
10    return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16    int result;
17
18    result = num1 + num2
19
20    return result;
21 }
```



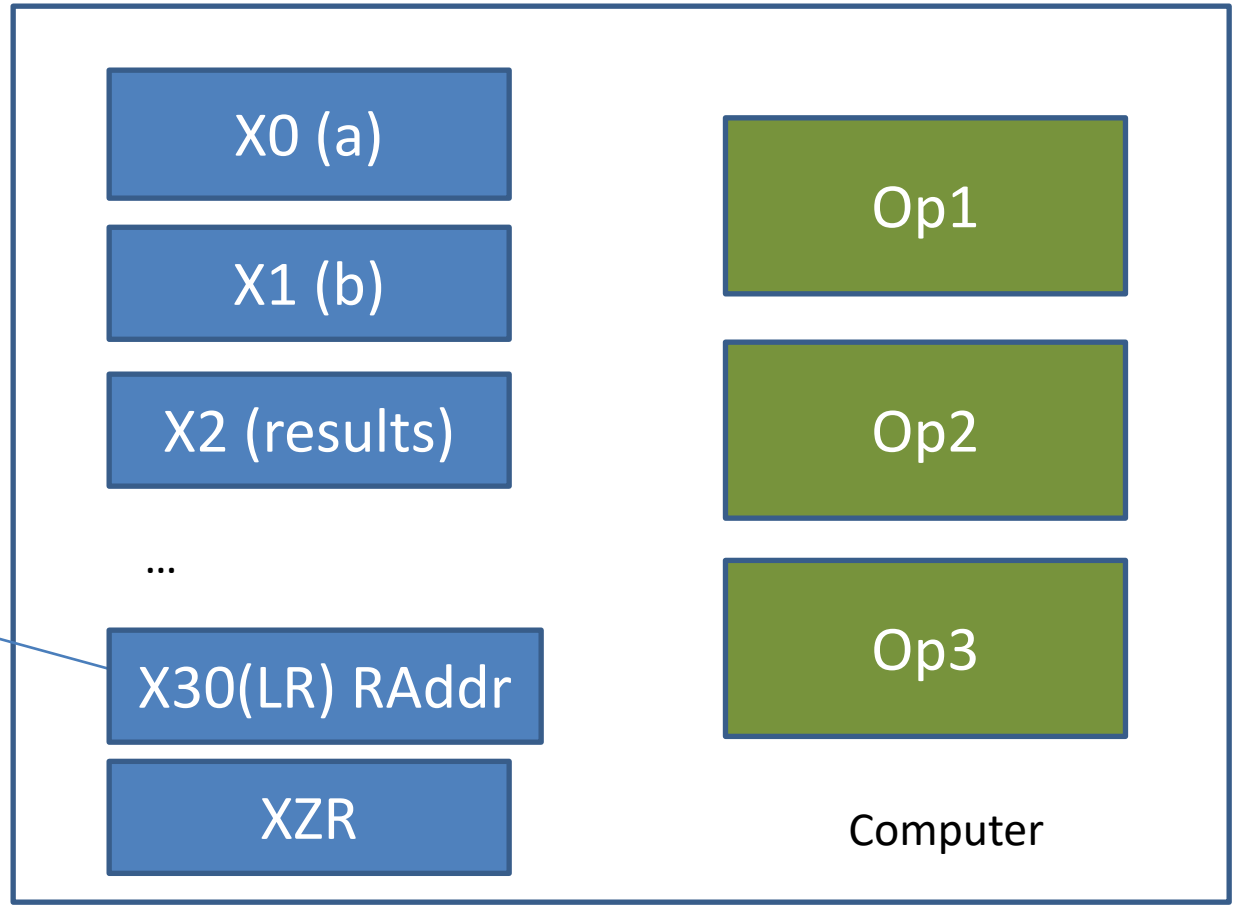
5. Put results in a register

What Registers used?

- X0 – X7: procedure **arguments/results**
- X30 (LR): link register (return address)
  - Also called as program counter (PC)
  - Program Counter: Contains the address of the current instruction

Steps in Executing a Procedure

```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



6. Return control to Caller

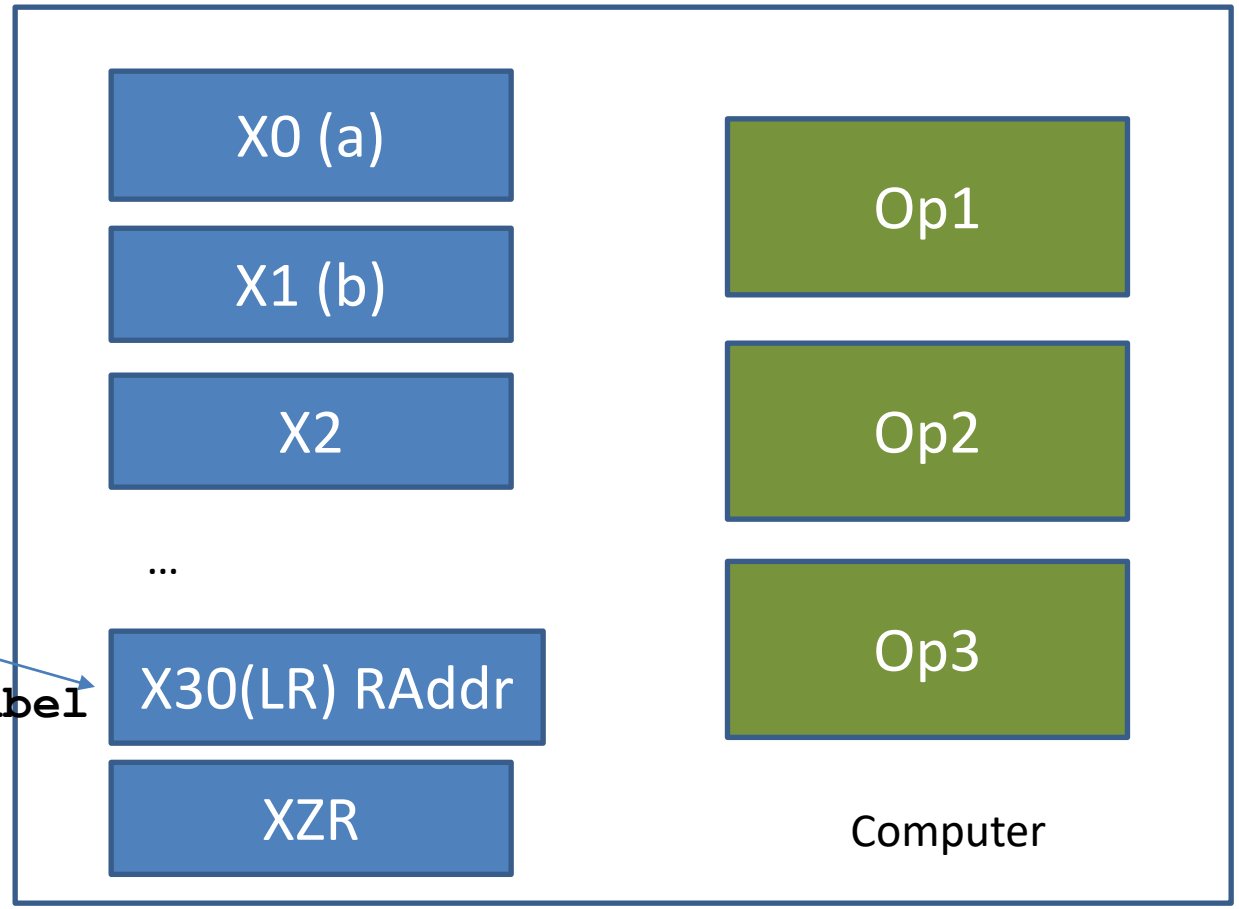
Procedure Calling

- 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

Steps in Executing a Procedure

```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int ret;
7  ---------------------------------
8      ret = add(a, b);
9
10     return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16     int result;
17
18     result = num1 + num2
19
20     return result;
21 }
```



BL ProcedureLabel

1. Caller puts parameters in registers for the procedure
    1. Specify a return address

Procedure Instructions

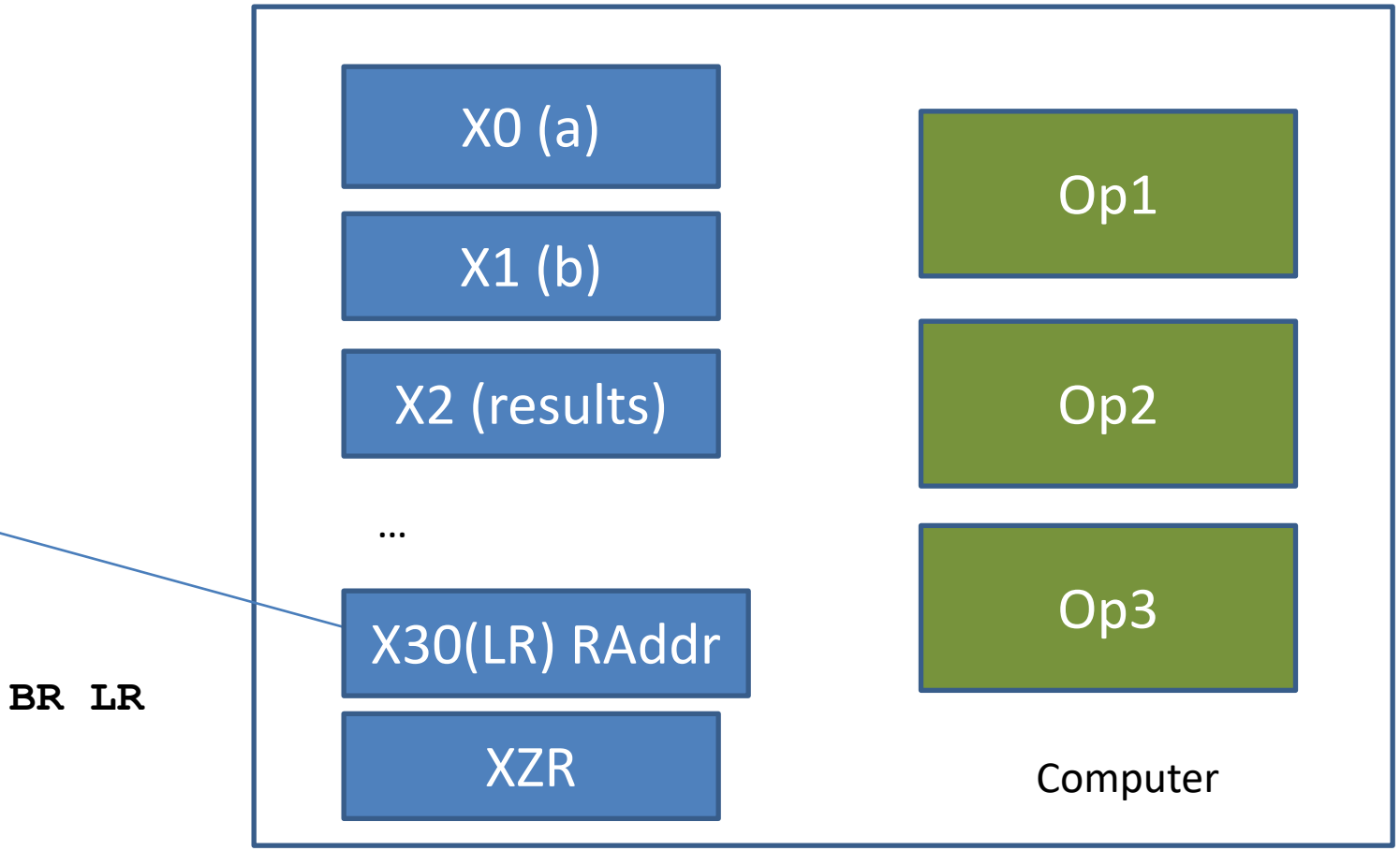
- Procedure call: jump and link

  `BL ProcedureLabel`

  – Address of following instruction put in X30 (LR)
    - Actually PC + 4 (32 bit instruction)
  – Jumps to target address

Steps in Executing a Procedure

```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int ret;
7
8     ret = add(a, b);
9
10    return 0;
11 }
12
13
14 int add(int num1, int num2) {
15
16    int result;
17
18    result = num1 + num2
19
20    return result;
21 }
```



BR LR

6. Return control to Caller

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

Instructions

Type	Name
Arithmetic	ADD, SUB, MUL
Data transfer	LDUR, STUR
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**

What if > 8 registers are needed by Callee?

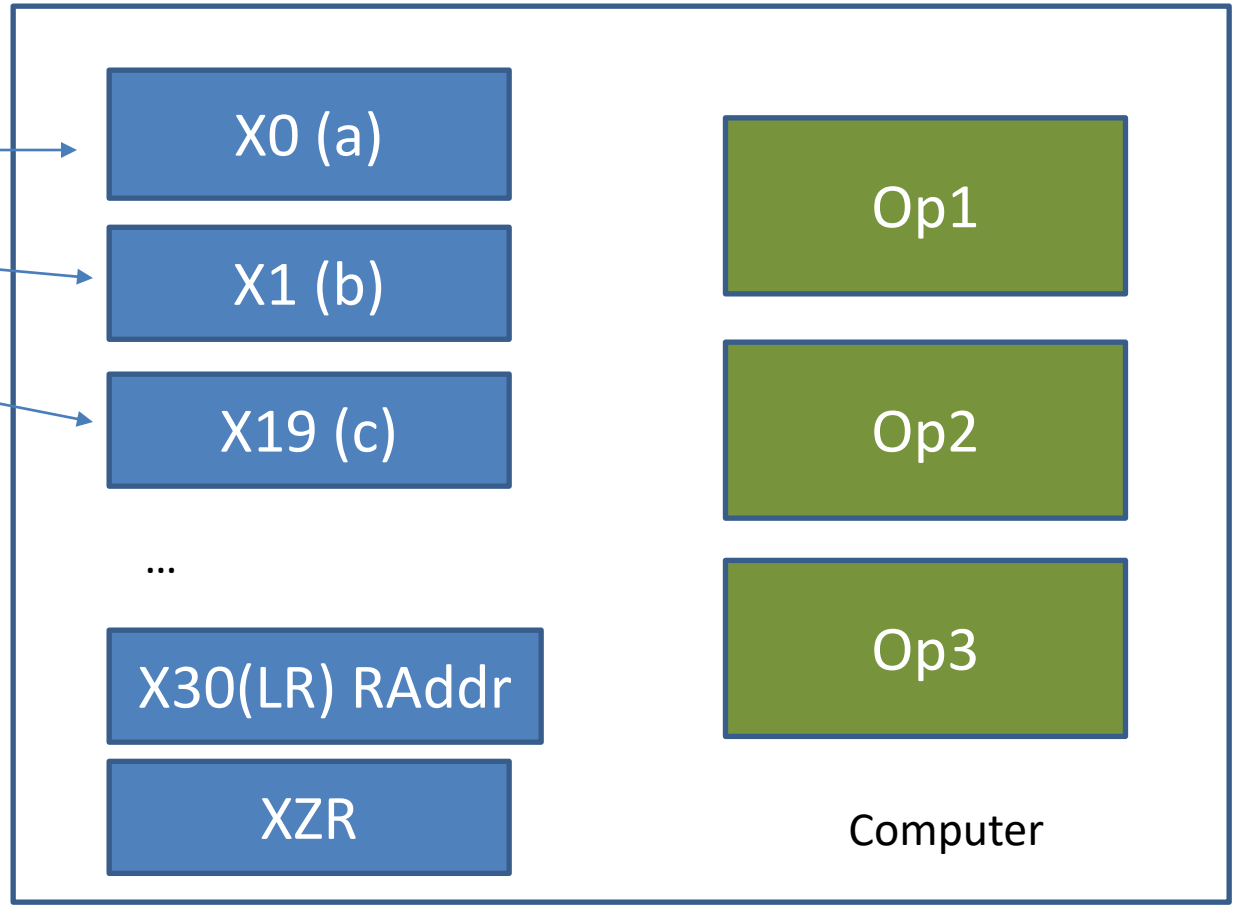
```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int c = 300
7     int ret;
8
9     ret = add(a, b);
10    c = c + ret
11    return 0;
12 }
13
14
15 int add(int num1, int num2) {
16
17    int result;
18    ... // More processing
19    result = num1 + num2
20
21    return result;
22 }
```

What if > 8 registers are needed by Callee?

```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int c = 300
7     int ret;
8
9     ret = add(a, b);
10    c = c + ret
11    return 0;
12 }
13
14
15 int add(int num1, int num2) {
16
17    int result;
18    ... // More processing
19    result = num1 + num2
20
21    return result;
22 }
```

What if > 8 registers are needed by Callee?

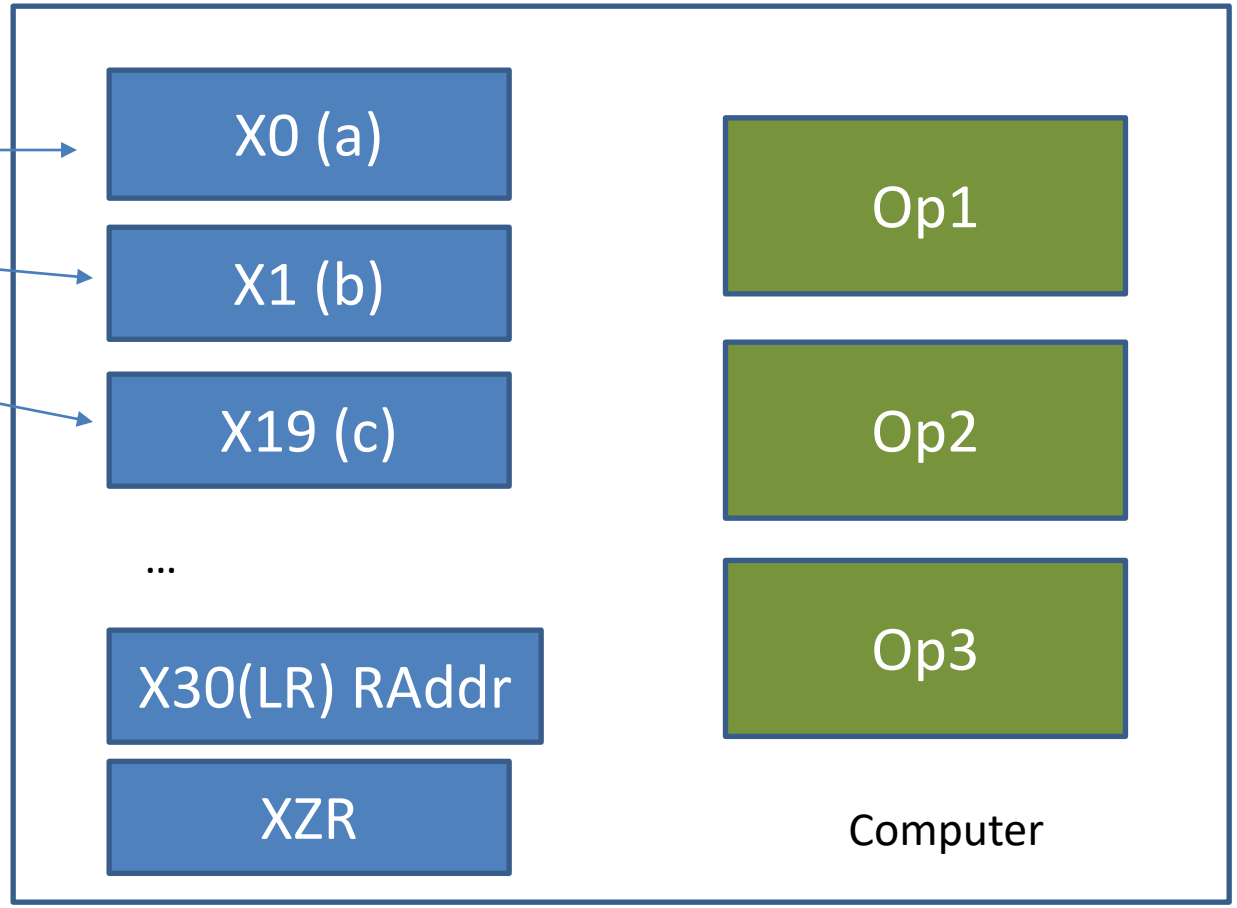
```
2  int main () {
3
4      int a = 100;
5      int b = 200;
6      int c = 300;
7      int ret;
8
9      ret = add(a, b);
10     c = c + ret;
11     return 0;
12 }
13
14
15 int add(int num1, int num2) {
16
17     int result;
18     ... // More processing
19     result = num1 + num2
20
21     return result;
22 }
```



Needs more registers?

What if > 8 registers are needed by Callee?

```
2  int main () {
3
4     int a = 100;
5     int b = 200;
6     int c = 300
7     int ret;
8
9     ret = add(a, b);
10    c = c + ret
11    return 0;
12 }
13
14
15 int add(int num1, int num2) {
16
17    int result;
18    ... // More processing
19    result = num1 + num2
20
21    return result;
22 }
```



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)

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

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?

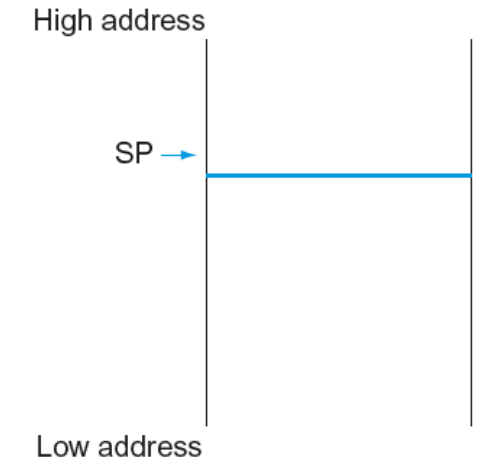
- X0 – X7: procedure arguments/results
- **X28 (SP): stack pointer (address of the most recently allocated stack)**
- X30 (LR): link register (return address)
  - Also called as program counter (PC)

Spilling to stack

- To spill registers (X10, X9, X19) on to the stack

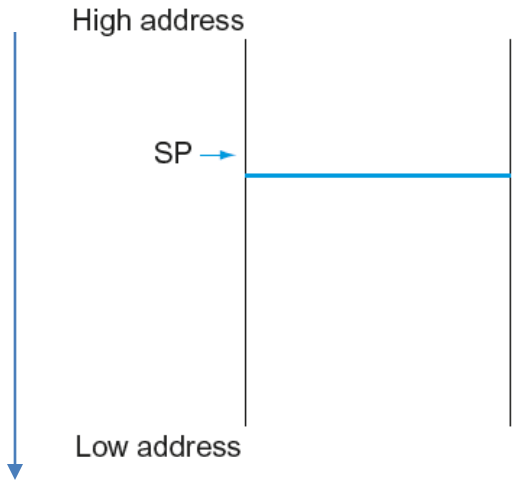
Spilling to stack

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



Spilling to stack

- To spill registers (X10, X9, X19) on to the stack
- Address of stack is save in X28 (SP)
- Goes from High to low for historic reasons

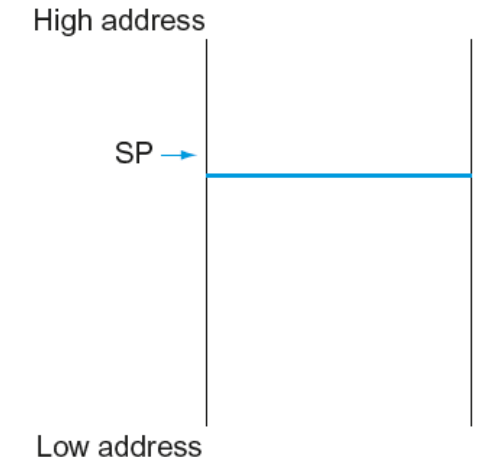


Spilling to stack

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

LEGv8 Code:

Make room for three items

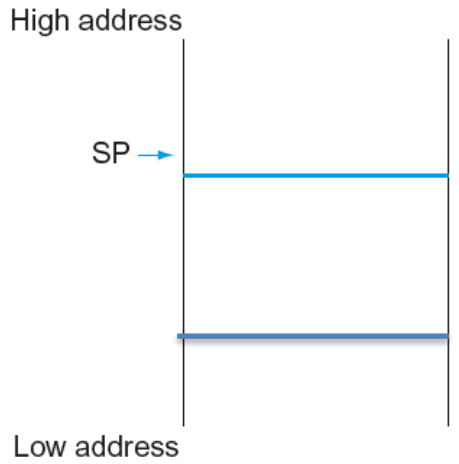


Spilling to stack

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

LEGv8 Code:

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



Spilling to stack

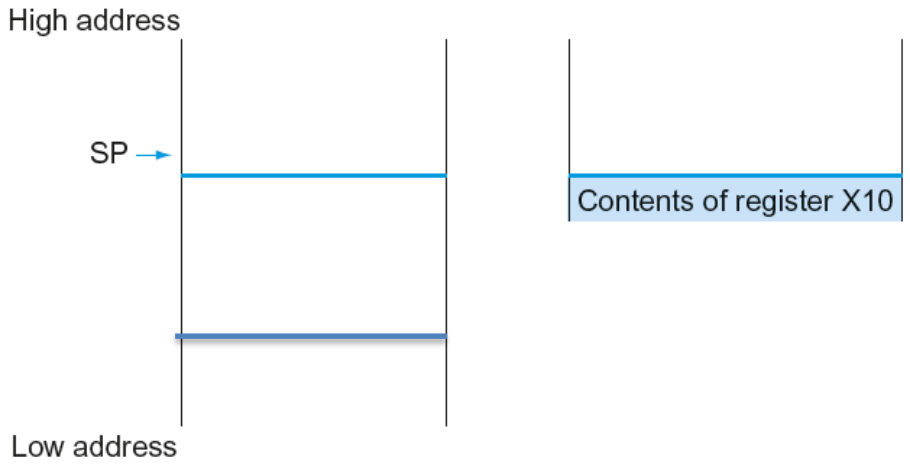
- To spill registers (X10, X9, X19) on to the stack
- Address of stack is save in X28 (SP)

LEGv8 Code:

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

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

(PUSH)

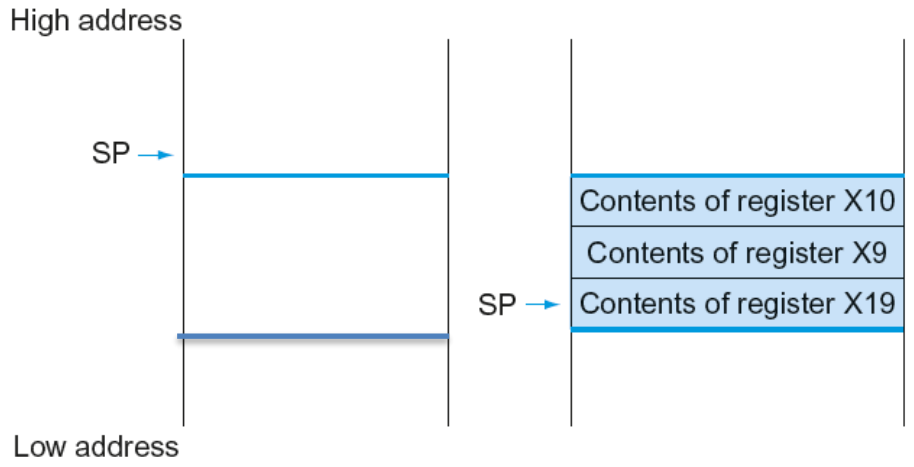


Spilling to stack

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

LEGv8 Code:

$SUBI\ SP, SP, \#24\ //\ Make\ room\ for\ three\ items$
$STUR\ X10, [SP, \#16]//Spill\ X10$
$STUR\ X9, [SP, \#8]//Spill\ X9$
$STUR\ X19, [SP, \#0]//Spill\ X19$



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)***

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

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

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