Chapter 2: Instructions: Language of the Computer

2.8 – 2.10 Procedure, String and Addressing for Wide

ITSC 3181 Introduction to Computer Architecture

Fall 2021

https://passlab.github.io/ITSC3181/

Department of Computer Science

Yonghong Yan

yyan7@uncc.edu

https://passlab.github.io/yanyh/

Chapter 2: Instructions: Language of the Computer

- Lecture
  - 2.1 Introduction
  - 2.2 Operations of the Computer Hardware
  - 2.3 Operands of the Computer Hardware
- Lecture
  - 2.4 Signed and Unsigned Numbers
  - 2.5 Representing Instructions in the Computer
- Lecture
  - 2.6 Logical Operations
  - 2.7 Instructions for Making Decisions

☛ Lecture
  - 2.8 Supporting Procedures in Computer Hardware
  - 2.9 Communicating with People
  - 2.10 RISC-V Addressing for Wide Immediate and Addresses
- Lecture
  - ~~2.11 Parallelism and Instructions: Synchronization~~
  - ~~2.12 Translating and Starting a Program~~
    - We covered before along with C Basics
  - 2.13 A C Sort Example to Put It All Together
  - 2.14 Arrays versus Pointers
    - We covered most before along with C Basics
  - ~~2.15 Advanced Material: Compiling C and Interpreting Java~~
  - 2.16 Real Stuff: MIPS Instructions
  - 2.17 Real Stuff: x86 Instructions
  - ~~2.18 Real Stuff: The rest of RISC-V~~
  - ~~2.19 Fallacies and Pitfalls~~
  - 2.20 Concluding Remarks
  - ~~2.21 Historical Perspective and Further Reading~~

Three Classes of Instructions We Will Focus On:

1. **Arithmetic-logic instructions**
   - add, sub, addi, and, or, shift left|right, etc

2. **Memory load and store instructions**
   - lw and sw: Load/store word
   - ld and sd: Load/store doubleword

3. **Control transfer instructions (changing sequence of instruction execution)**
   - Conditional branch: bne, beq
   - Unconditional jump: j
   - **Procedure call and return: jal and jr**

Procedure Call: sum_full.c

sum_full.c

```
35 REAL sum(int N, REAL X[], REAL a) {
36     int i;
37     REAL result = 0.0;
38     for (i = 0; i < N; ++i)
39         result += a * X[i];
40     return result;
41 }
```

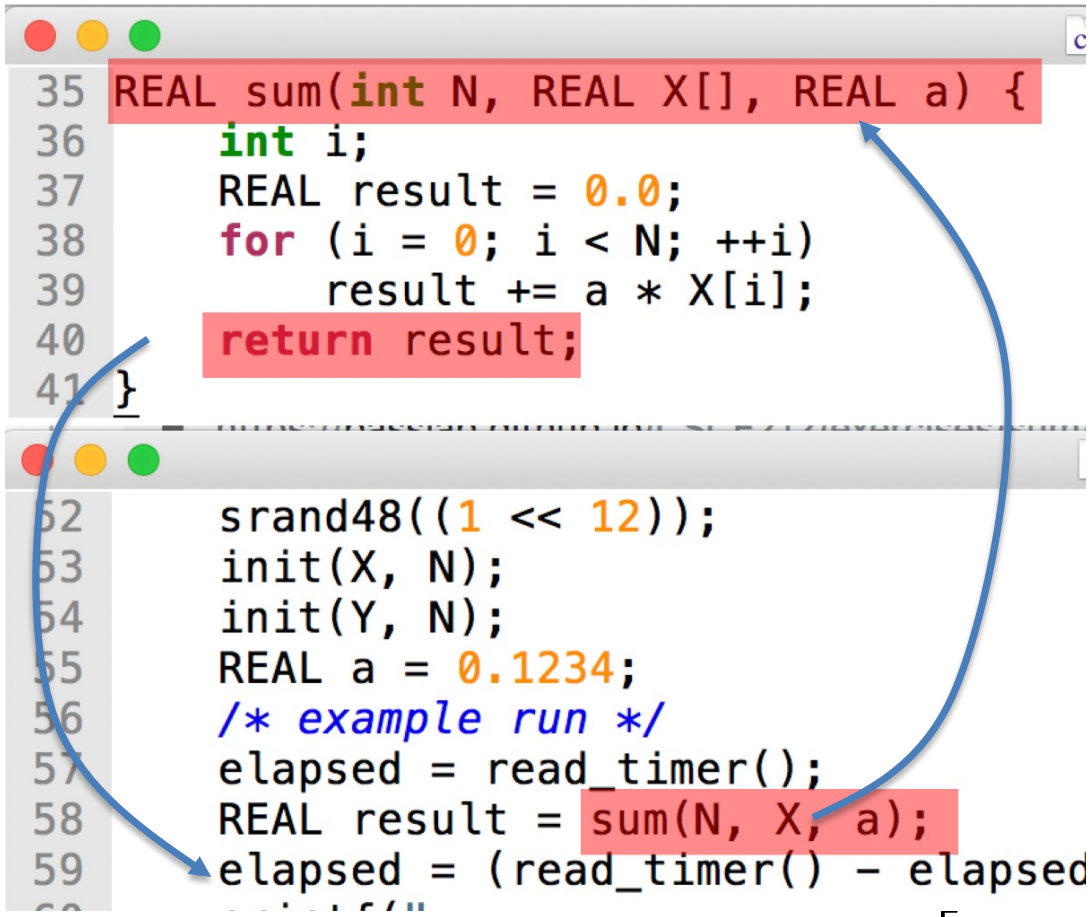
sum_full.c

```
52     srand48((1 << 12));
53     init(X, N);
54     init(Y, N);
55     REAL a = 0.1234;
56     /* example run */
57     elapsed = read_timer();
58     REAL result = sum(N, X, a);
59     elapsed = (read_timer() - elapsed);
```

https://passlab.github.io/ITSC3181/exercises/sum

Procedure Call

- Control is transferred when there is procedure call and return
- Steps required
  1. Place parameters in registers
  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



Sum Example, sum_full_riscv.s

```
    return result;
```

```
 96         .globl  sum
 97         .type   sum, @function
 98 sum:
 99         addi    sp,sp,-48
100         sd      s0,40(sp)
101         addi    s0,sp,48
102         mv      a5,a0
103         sd      a1,-48(s0)
104         fsw     fa0,-40(s0)
105         sw      a5,-36(s0)
127         sext.w  a5,a5
128         blt     a4,a5,.L10
129         flw     fa5,-24(s0)
130         fmv.s   fa0,fa5
131         ld      s0,40(sp)
132         addi    sp,sp,48
133         jr      ra
134
```

Return to caller with return value stored in register

```
REAL result = sum(N, X, a);
```

```
156         .globl  main
157         .type   main, @function
158 main:
159         addi    sp,sp,-80
160         sd      ra,72(sp)
161         sd      s0,64(sp)
162         addi    s0,sp,80
163         mv      a5,a0
215         fsw     fa5,-44(s0)
216         call    read_timer
217         fsd     fa0,-56(s0)
218         lw      a5,-20(s0)
219         flw     fa0,-44(s0)
220         ld      a1,-32(s0)
221         mv      a0,a5
222         call    sum
223         fsw     fa0,-6
224         call    read_timer
```

Args for sum call

Store return address reg and call transfer sum

https://passlab.github.io/ITSC3181/exercises/sum

Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  – Address of following instruction put in x1

  – Jumps to target address

- Procedure return: jump and link register

  `jalr x0, 0(x1)`

  – Like jal, but jumps to 0 + address in x1

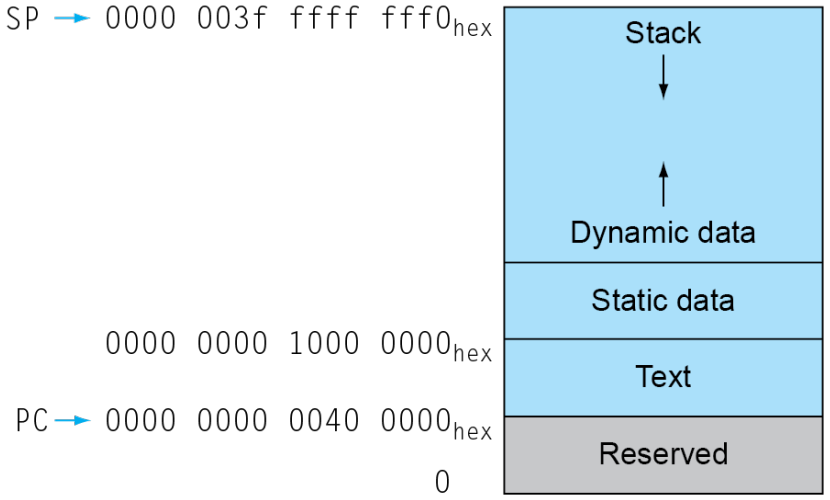
  – Use x0 as rd (x0 cannot be changed)

  – Can also be used for computed jumps

    - e.g., for case/switch statements

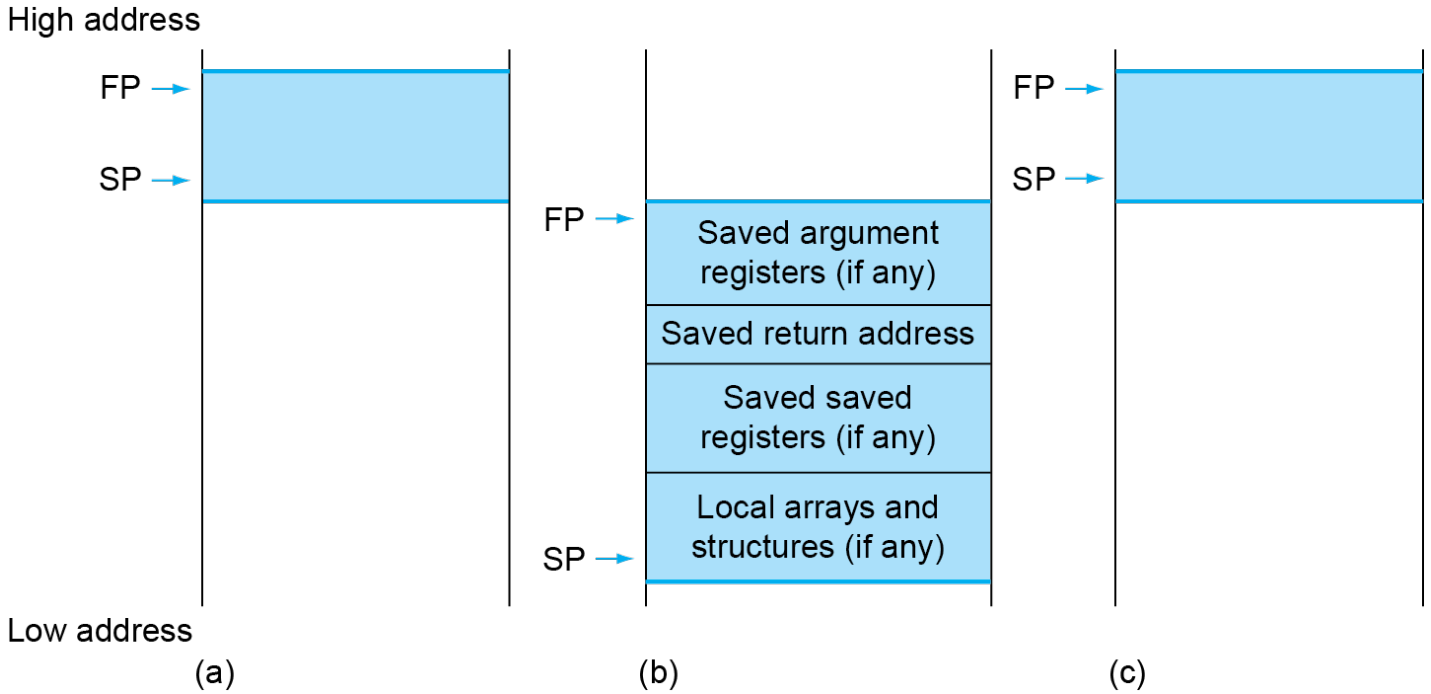
Memory Layout of a Process

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



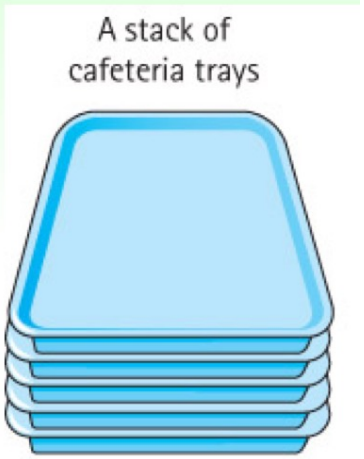
Local Data on the Stack

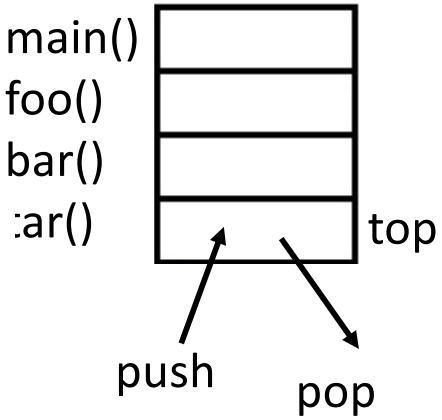
- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

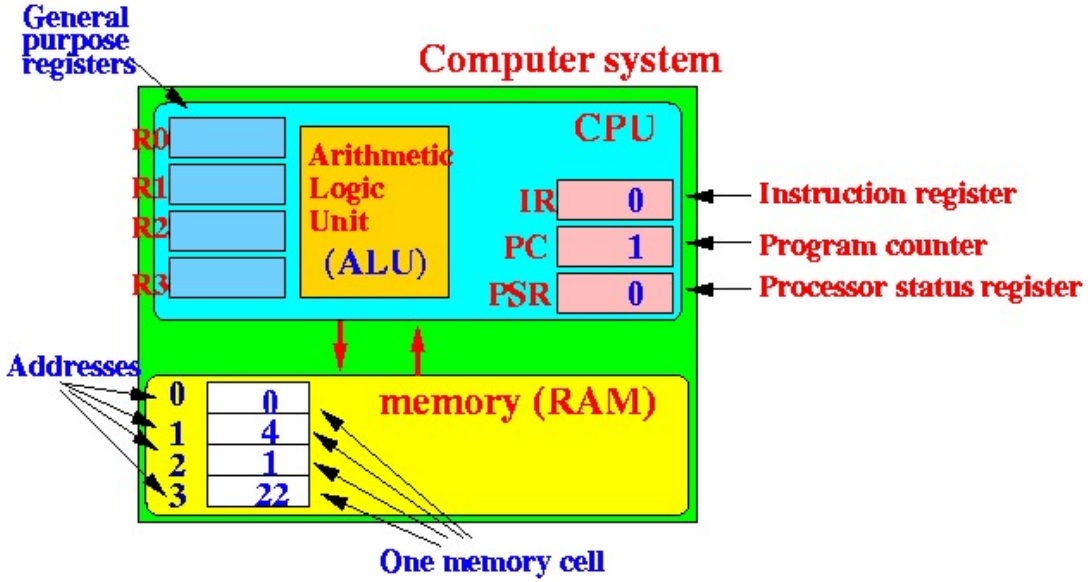


Stack Memory Used for Function Calls

- Stack is Last-In-First-Out (LIFO) data structure to store the info of each function of the call path
  - Main() calls foo(), foo() calls bar(), bar() calls tar()
  - Call in: push function to the stack top
  - Return: pop function from the top
- Stack frame, function frame, activation record
  - The memory and the data of the info for each function call

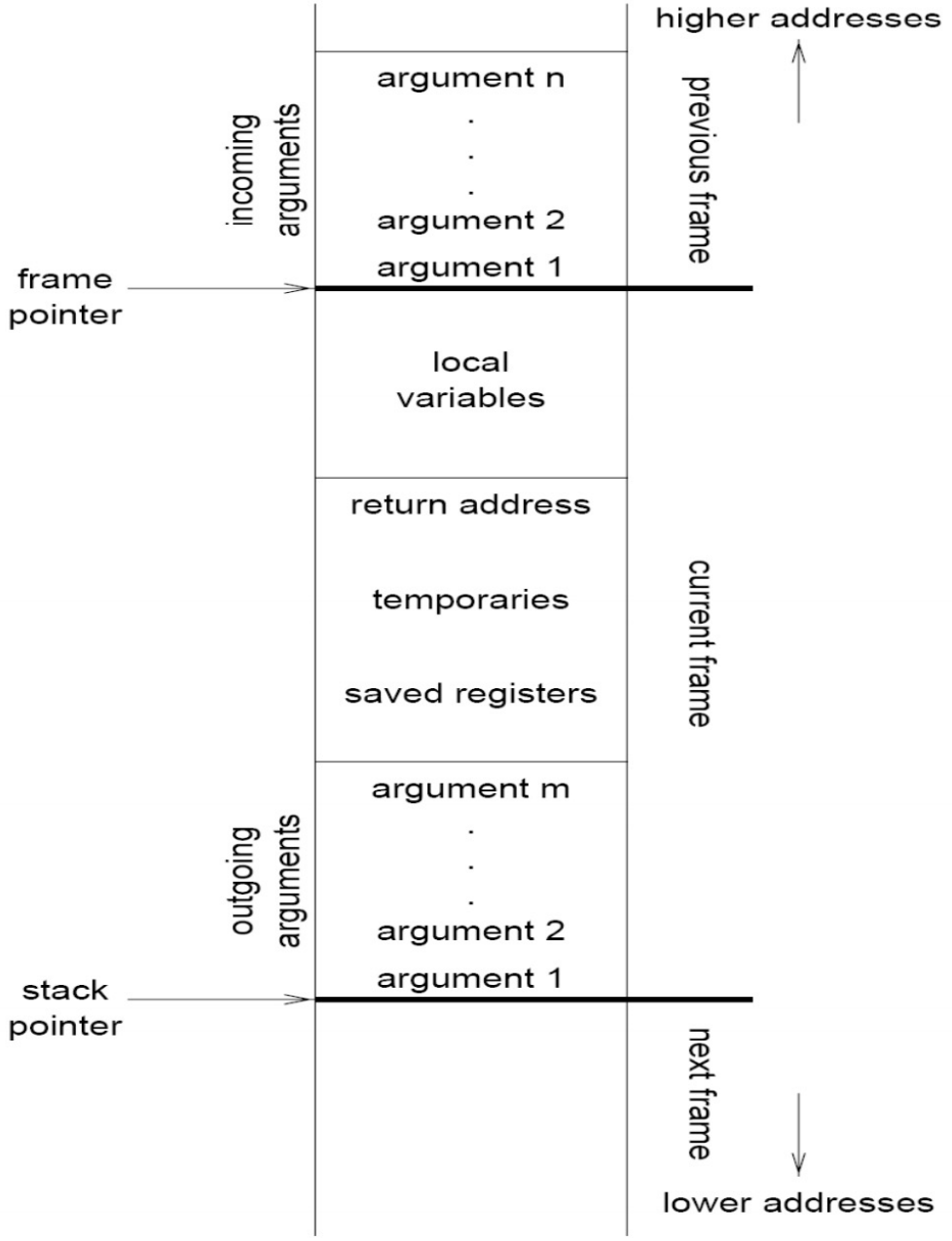






Stack Frame (Activation Record) of a Function Call

- Information:
  - Parameters
  - Local variables
  - Return address
  - Location to put return value when function exits
  - Control link to the caller's activation record
  - Saved registers
  - Temporary variables and intermediate results
  - (not always) Access link to the function's static parent
- Frame pointer (fp register): the starting address of AR
- Stack pointer (sp register): the ending address of AR



Leaf Procedure Example

- Leaf procedure
  - A procedure does not call other procedures
    - Thinking of procedure calls as a tree
- 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;
}
```

  - Arguments g, …, j in x10, …, x13
  - **f in x20**
  - **temporaries x5, x6**
  - Need to save x5, x6, x20 on stack

Local Data on the Stack

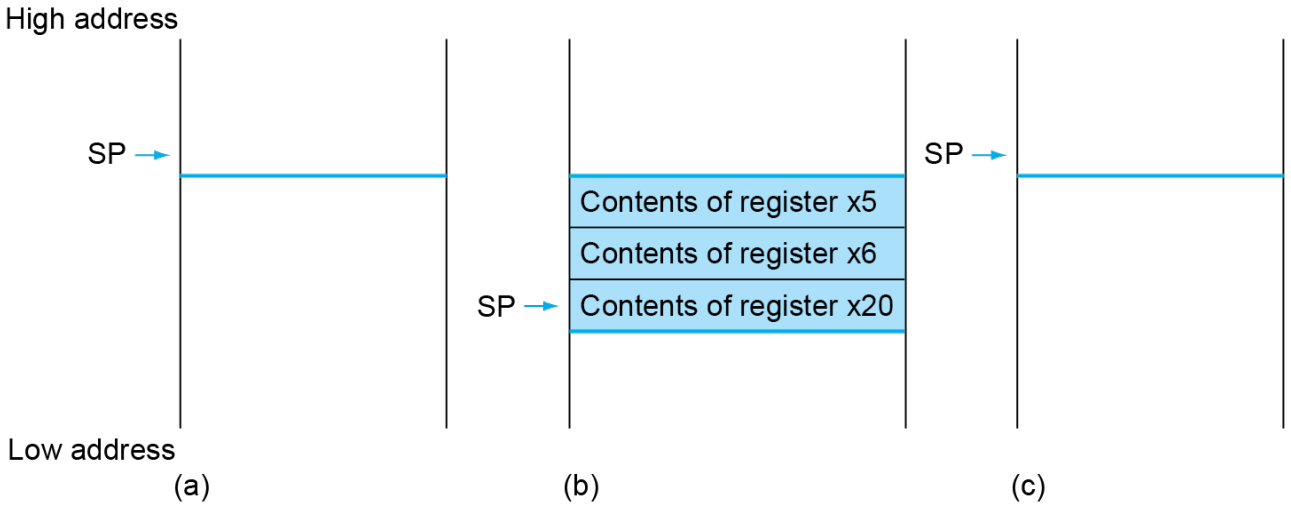
- Stack before, during and after the function call
  - SP (stack pointer) is the register that store the address of the current function call

```
int main ( … ) {
  …
  long long int result = leaf_example( … );
  …
}
```

```
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;
}
```

- Arguments g, …, j in x10, …, x13
- **f in x20**
- **temporaries x5, x6**
- Need to save x5, x6, x20 on stack

If caller uses x5, x6 or x20, we have to preserve them. They are preserved in the callee stack.



Leaf Procedure Example

```
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;
}
```

- Arguments g, …, j in x10, …, x13
- **f in x20**
- **temporaries x5, x6**
- Need to save x5, x6, x20 on stack

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24          Adjust stack to make room for 3 items
  sd   x5,16(sp)          Save x5, x6, x20 on stack
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11         x5 = g + h
  add  x6,x12,x13         x6 = i + j
  sub  x20,x5,x6          f = x5 – x6
  addi x10,x20,0          copy f to return register
  ld   x20,0(sp)          Resore x5, x6, x20 from stack
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24           Adjust back to return memory for 3 items
  jalr x0,0(x1)           Return to caller
```

Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not automatically preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

FIGURE 2.14 RISC-V register conventions.

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

  - **Argument n in x10**
  - **Result in x10**

- It is a recursive function.

Leaf Procedure Example

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

- **Argument n in x10**
- **Result in x10**

- RISC-V code:

```
fact:
    addi sp,sp,-16          Adjust stack for two items
    sd   x1,8(sp)           Save return address and n on stack
    sd   x10,0(sp)          Save the argument n
    addi x5,x10,-1          x5 = n - 1
    bge  x5,x0,L1           if n >= 1, go to L1
    addi x10,x0,1           Else, set return value to 1
    addi sp,sp,16           Pop stack, don't bother restoring values
    jalr x0,0(x1)           Return
L1: addi x10,x10,-1         n = n - 1
    jal  x1,fact            call fact(n-1)
    addi x6,x10,0           move result of fact(n - 1) to x6
    ld   x10,0(sp)          Restore caller's n
    ld   x1,8(sp)           Restore caller's return address
    addi sp,sp,16           Pop stack
    mul  x10,x10,x6         return n * fact(n-1)
    jalr x0,0(x1)           return
```

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters

- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

ASCII Characters

- Each character is represented by a 8-bit byte ➔ max 256

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters.

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - A string is an array of characters with` \0` as the last character
    - char x[100]; a string of 100 character
    - char * x2; is used for refer to a string
  - Null-terminated string

```
void strcpy (char x[], char y[])
  { size_t i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
```

  - **Base address for x and y are in x10 and x11**
  - **i is in x19**

String Copy Example

```
void strcpy (char x[], char y[])
  { size_t i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
```

- RISC-V code:

```
strcpy:
    addi sp,sp,-8          // adjust stack for 1 doubleword
    sd   x19,0(sp)         // save x19
    add  x19,x0,x0         // i=0
L1: add  x5,x19,x10        // x5 = addr of y[i]
    lbu  x6,0(x5)          // x6 = y[i]
    add  x7,x19,x10        // x7 = addr of x[i]
    sb   x6,0(x7)          // x[i] = y[i]
    beq  x6,x0,L2          // if y[i] == \0 then exit
    addi x19,x19,1         // i = i + 1
    jal  x0,L1             // next iteration of loop
L2: ld   x19,0(sp)         // restore saved x19
    addi sp,sp,8           // pop 1 doubleword from stack
    jalr x0,0(x1)          // and return
```

32-bit Constants

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Most constants are small
  - I-format instructions have only 12 bits for immediate
    - E.g. addi x6, x0, 1024
  - 12-bit immediate is sufficient most of the time
- For the occasional 32-bit constant

  `lui rd, constant (load upper immediate)`
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

`lui x19, 976  // 0x003D0`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

`addi x19,x19,128  // 0x500`

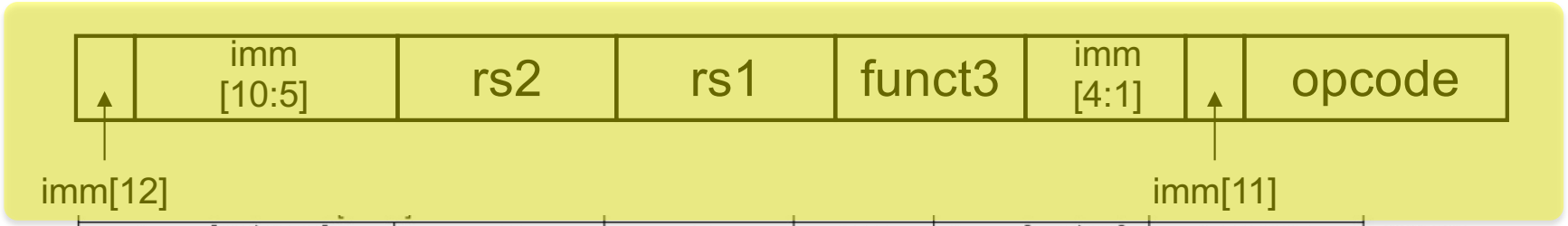
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Now x19 has $976*2^{12} + 128$

SB-Format Encoding for Branch Instr (e.g. beq)

http://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf#page=116

- Branch instructions, e.g. ”beq x3, x4, EXIT”, specify
  - Opcode, two source registers (rs1 and rs2), target address as imm
  - **Most branch targets are near branch, Forward or backward**
- SB-Format instructions: beq x8, x9, **4**



imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

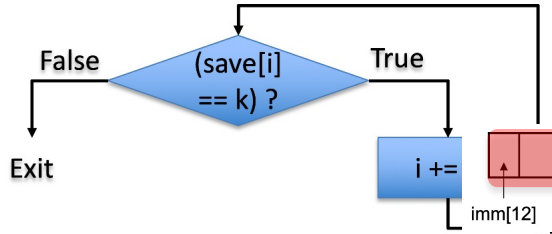
- **PC-relative addressing**
  - **Branch target address is encoded as the offset off the the address of the branch instruction itself**
  - **Target address = PC (Branch address) + immediate × 2**

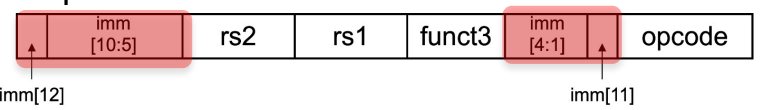
imm of beq instruction is the offset between the address of beq instruction and the target address (page 115 of the text book)

- C code:

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

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





- RISC-V code: (save[i] is to be read/loaded)

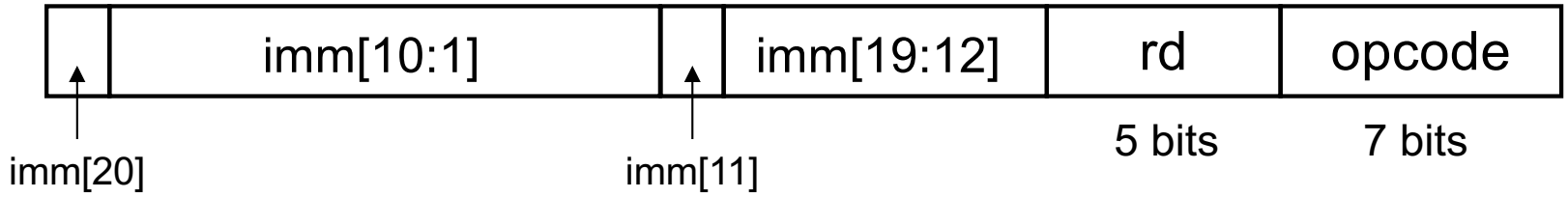
```
Loop: slli x10, x22, 3   //x10 has i*8
      add  x10, x10, x25 //base+offset
      ld   x9,  0(x10)//newbase in x10
      bne  x9,  x24, Exit //false
      addi x22, x22, 1 //true,the loop body,i=i+1
      beq  x0,  x0,  Loop
Exit: …
```

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

- The Exit offset of the bne is encoded as 6 (..0110)
  – Offset is 6*2 = 12 bytes, i.e. 3 instr forward
  – Exit's address = bne's address (80012) + 12 = 80024 (Exit)
- The Loop offset of the beq is encoded as -10 (..110110)
  – Offset is -10*2 = -20 bytes, i.e. 5 instr backward
  – Loop's address = beq's address (80020) + -20 = 80000 (Loop)
- imm[11] is the sign bit of the imm → help decoding
- To calculate the imm for beq: (target-PC)/2

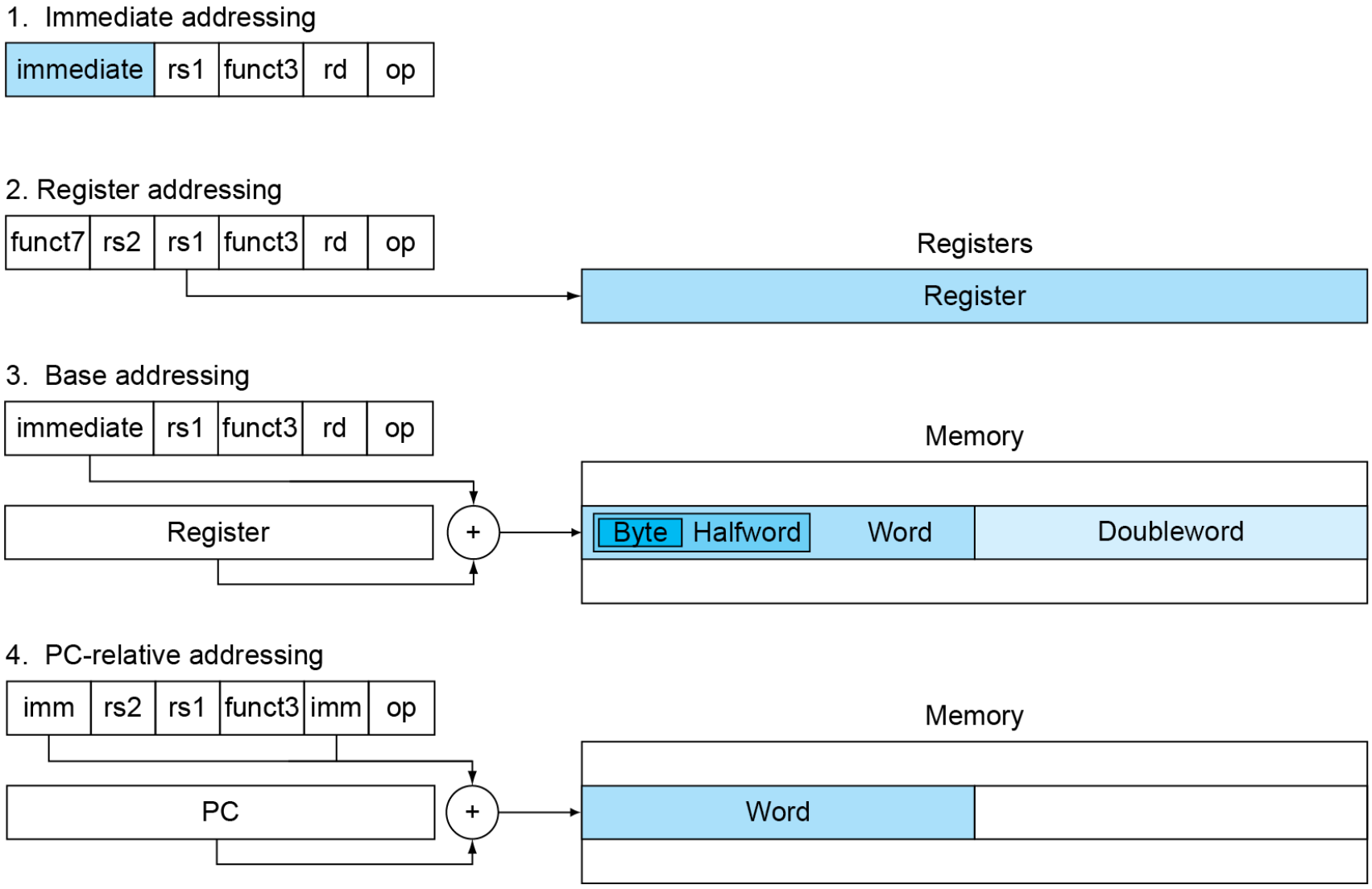
Jump Addressing

- Jump and link (`jal`) target uses 20-bit immediate for larger range
- UJ format:



  - For long jumps, eg, to 32-bit absolute address
    - lui: load address[31:12] to temp register
    - jalr: add address[11:0] and jump to target

Summary of RISC-V Addressing (How Operands are Specified or Provided)



RISC-V Encoding and Decoding: Encoding Format

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

RISC-V Encoding and Decoding: Opcode/Funct

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
R-type	funct7	rs2	rs1	funct3	rd	opcode
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode
U-type	immediate[31:12]				rd	opcode

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	iw	0000011	010	n.a.
	id	0000011	011	n.a.
	ibu	0000011	100	n.a.
	ihu	0000011	101	n.a.
	iwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

To Decode an Instruction Word

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- To decode an instruction word: 00578833_{hex}
  1. Convert to binary: 0000 0000 0101 0111 1000 1000 0011 0011
  2. Determine the opcode, the rightmost 7 bits: 011 0011
  3. It is R-type arithmetic instruction
  4. Decode the rest, func3 and funct7 and then rs1, rs2, rd

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

  5. Add x16, x15, x5