SimpleRISC ISA

SimpleRisc

* Simple RISC ISA
* Contains only 21 instructions
* We will design an assembly language for SimpleRisc
* Design a simple binary encoding,
* and then implement it ...

Mc Graw Hill Education

Survey of Instruction Sets

ISA	Type	Year	Vend	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPA	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64		32
RC PowerPC	RISC	1992	Apple,IBM,Motorola	32		32
	RISC	2002	Apple,IBM	64		32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32		32
	RISC	1999	MIPS	64		32
Alpha	RISC	1992	DEC	64		32
x86	CISC	1978	Intel,AMD	16	little	
	CISC	1985	Intel,AMD	32	little	
	CISC	2003	Intel,AMD	64	little	16
ARM	RISC	1985	ARM	32	bi(little default)	16
	RISC	2011	ARM	64	bi(little default)	31

Registers

* SimpleRisc has 16 registers
  * Numbered : r0 … r15
  * r14 is also referred to as the stack pointer (sp)
  * r15 is also referred to as the return address register (ra)
* View of Memory
  * Von Neumann model
  * One large array of bytes
* Special flags register → contains the result of the last comparison

mov instruction

mov r1,r2	$r1 \leftarrow r2$
mov r1,3	$r1 \leftarrow 3$

* Transfer the contents of one register to another
* Or, transfer the contents of an immediate to a register
* The value of the immediate is embedded in the instruction
  * SimpleRisc has 16 bit immediates
  * Range -2^{15} to $2^{15} - 1$

Arithmetic/Logical Instructions

* SimpleRisc has 6 arithmetic instructions
  * add, sub, mul, div, mod, cmp

Example	Explanation
add r1, r2, r3	$r1 \leftarrow r2 + r3$
add r1, r2, 10	$r1 \leftarrow r2 + 10$
sub r1, r2, r3	$r1 \leftarrow r2 - r3$
mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
div r1, r2, r3	$r1 \leftarrow r2/r3$ (quotient)
mod r1, r2, r3	$r1 \leftarrow r2$ *mod* $r3$ (remainder)
cmp r1, r2	set flags

Examples of Arithmetic Instructions

* Convert the following code to assembly

```
a = 3
b = 5
c = a + b
d = c - 5
```

* Assign the variables to registers
  * a ← r0, b ← r1, c ← r2, d ← r3

```
mov r0, 3
mov r1, 5
add r2, r0, r1
sub r3, r2, 5
```

Examples - II

* Convert the following code to assembly

```
a = 3
b = 5
c = a  * b
d = c mod 5
```

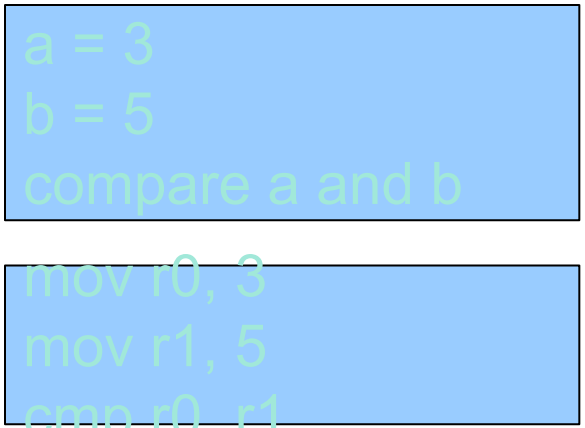
* Assign the v s

```
mov r0, 3
mov r1, 5
mul r2, r0, r1
mod r3, r2, 5
```

  * a ← r0, b ←

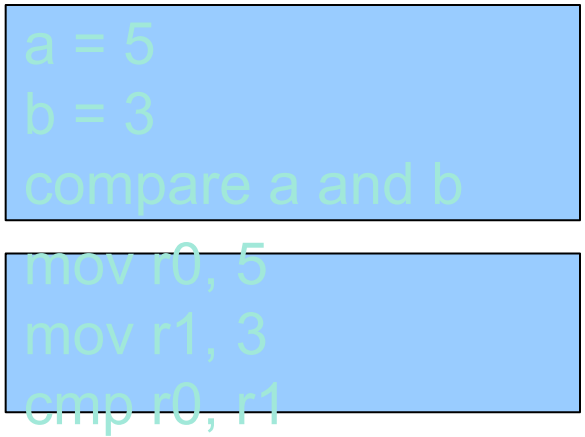
Compare Instruction

* Compare 3 and 5, and print the value of the flags



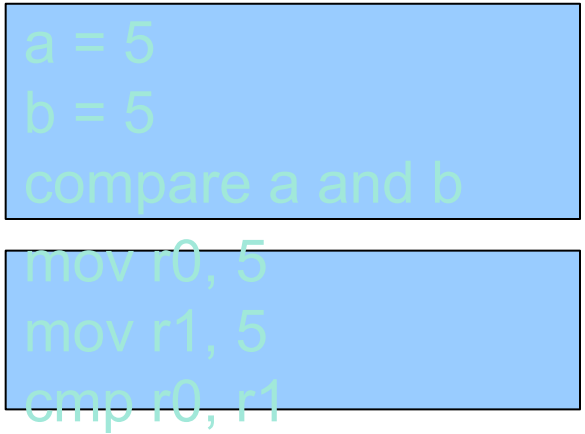
Compare Instruction

* Compare 5 and 3, and print the value of the flags



Compare Instruction

* Compare 5 and 5, and print the value of the flags



Example with Division

Write assembly code in SimpleRisc to compute: 31 / 29 - 50, and save the result in r4.

Answer:

```
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

SimpleRisc, r1

Logical Instructions

and r1, r2, r3	$r1 \leftarrow r2 \,\&\, r3$
or r1, r2, r3	$r1 \leftarrow r2 \mid r3$
not r1, r2	$r1 \leftarrow \sim r2$

& bitwise AND, | bitwise OR, ~ logical complement

* The second argument can either be a register or an immediate

Compute (a | b). Assume that a is stored in r0, and b is stored in r1. Store the result in r2.

Answer:

Shift Instructions

* Logical shift left (lsl) (<< operator)
  * 0010 << 2 is equal to 1000
  * (<< n) is the same as multiplying by 2^n
* Arithmetic shift right (asr) (>> operator)
  * 0010 >> 1 = 0001
  * 1000 >> 2 = 1110
  * same as dividing a signed number by 2^n

Shift Instructions - II

* logical shift right (lsr) (>>> operator)
  * 1000 >>> 2 = 0010
  * same as dividing the unsigned representation by 2^n

Example	Explanation
lsl r3, r1, r2	$r3 \leftarrow r1 << r2$ (shift left)
lsl r3, r1, 4	$r3 \leftarrow r1 << 4$ (shift left)
lsr r3, r1, r2	$r3 \leftarrow r1 >>> r2$ (shift right logical)
lsr r3, r1, 4	$r3 \leftarrow r1 >>> 4$ (shift right logical)
asr r3, r1, r2	$r3 \leftarrow r1 >> r2$ (arithmetic shift right)
asr r3, r1, 4	$r3 \leftarrow r1 >> r2$ (arithmetic shift right)

Example with Shift Instructions

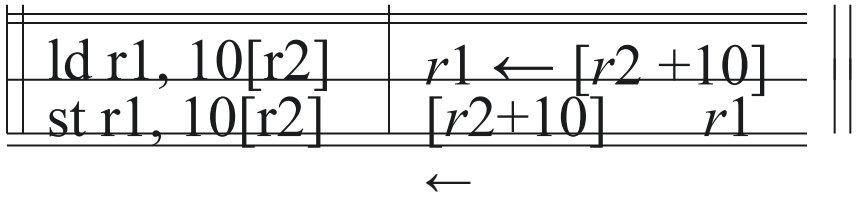
* Compute 101 * 6 with shift operators

Example with Shift Instructions

* Compute 101 * 6 with shift operators

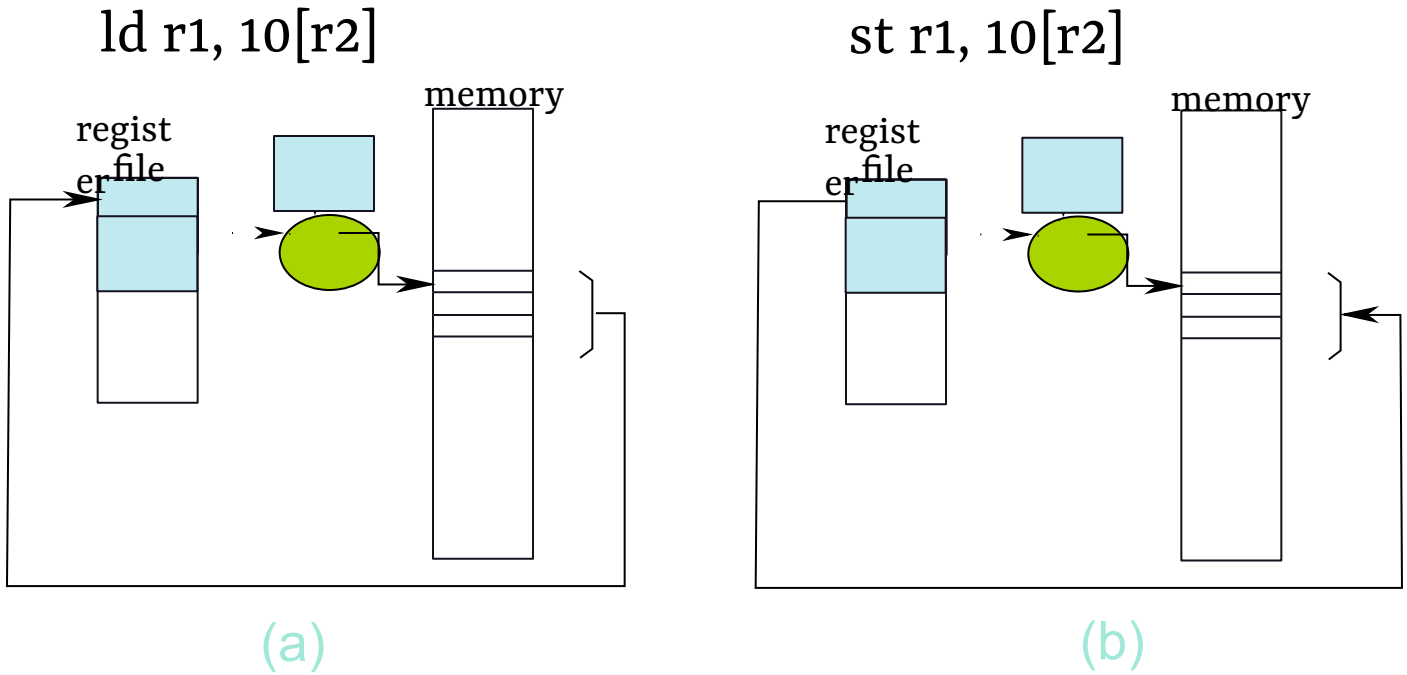
```
mov r0, 101
lsl r1, r0, 1
lsl r2, r0, 2
add r3, r1, r2
```

Load-store instructions



* 2 address format, base-offset addressing
* Fetch the contents of r2, add the offset (10), and then perform the memory access

Load-Store


ld r1, 10[r2]
memory
regist
er file
st r1, 10[r2]
memory
regist
er file
(a)
(b)

Example – Load/Store

* Translate :

```
int arr[10];
arr[3] = 5;
arr[4] = 8;
arr[5] = arr[4] + arr[3];
```

```
/* assume base of array saved in r0 */
mov r1, 5
st r1, 12[r0]
mov r2, 8
st r2, 16[r0]
add r3, r1, r2
st r3, 20[r0]
```

Branch Instructions

* Unconditional branch instruction

b .foo	branch to .foo

```
        add     r1, r2, r3
        b       .foo
        …
        …
.foo:
        add     r3, r1, r4
```

Conditional Branch Instructions

beq .foo	branch to .foo if *flags.E* = 1
bgt .foo	branch to .foo if *flags.GT* = 1

* The flags are only set by cmp instructions
* beq (branch if equal)
  * If flags.E = 1, jump to .foo
* bgt (branch if greater than)
  * If flags.GT = 1, jump to .foo

Examples

* If r1 > r2, then save 4 in r3, else save 5 in r3

Examples

* If r1 > r2, then save 4 in r3, else save 5 in r3

```
                cmp     r1, r2
                bgt     .gtlabel
                mov     r3, 5
                b .joinlabel
.gtlabel:
                mov     r3, 4
.joinlabel:     …
                ...
```

Example - II

```
        mov r1, 1
        mov r2, r0
.loop:
        mul r1, r1, r2
        sub r2, r2, 1
        cmp r2, 1
        bgt .loop
```

Example - II

Answer: Compute the factorial of the variable num.

C

```
int prod = 1;
int idx;
for(idx = num; idx > 1; idx --) {
        prod = prod * idx
}
```

SimpleRisc

```
            mov r1, 1                   /* prod = 1 */
            mov r2, r0                  /* idx = num */
.loop:
            mul r1, r1, r2      /* prod = prod * idx */
            sub r2, r2, 1       /* idx = idx - 1 */
            cmp r2, 1           /* compare (idx, 1) */
            bgt .loop           /* if (idx > 1) goto .loop*/
```

Exercise

You are given a number in register r1. You may assume that this number is greater than 3. Write a SimpleRISC program to find out if this number is a prime number or not. If it is, write 1 to r0. Else, write 0 to r0.

You are given a number in register r1. You may assume that this number is greater than 3. Write a SimpleRISC program to find out if this number is a prime number or not. If it is, write 1 to r0. Else, write 0 to r0.

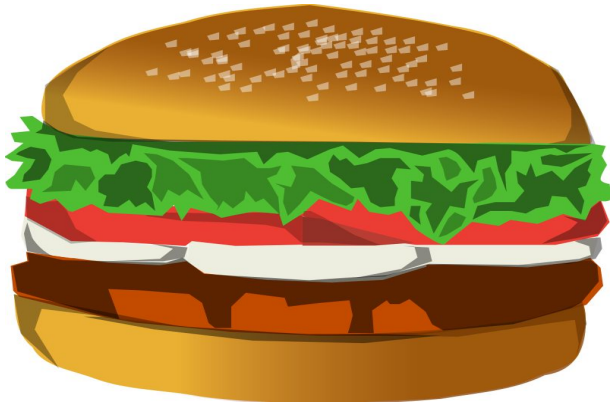
```
        mov r2, 2
.loop:
        mod r3, r1, r2    @ divide number by r2
        cmp r3, 0         @ compare the result with 0
        beq .notprime     @ if the result is 0, not prime
        add r2, r2, 1     @ increment r2
        cmp r1, r2        @ compare r2 with the number
        bgt .loop         @ iterate if r2 is smaller
        mov r0, 1         @ number is prime
        b .exit           @ exit
.notprime:
        mov r0, 0         @ number is not prime
.exit:
```

Exercise

You are given two numbers in registers r1 and r2. You may assume that these numbers are greater than 0. Write a SimpleRISC program to find out the LCM of these two numbers. Save the result in r0.

You are given two numbers in registers r1 and r2. You may assume that these numbers are greater than 0. Write a SimpleRISC program to find out the LCM of these two numbers. Save the result in r0.

```
@ let the numbers be A(r1) and B(r2)
        mov r3, 1           @ idx = 1
        mov r4, r1          @ L = A
.loop:
        mod r5, r4, r2      @ L = A % B
        cmp r5, 0           @ compare mod with 0
        beq .lcm            @ LCM found (L is the LCM)
        add r3, r3, 1       @ increment idx
        mul r4, r1, r3      @ L = A * idx
        b .loop
.lcm:
        mov r0, r4          @ result is equal to L
```



* Write a SimpleRisc assembly program to find the smallest number that is a sum of two cubes in two different ways → 1729

Modifiers

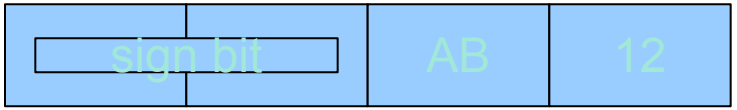
* We can add the following modifiers to an instruction that has an immediate operand
* Modifier :
  * default : mov → treat the 16 bit immediate as a signed number (automatic sign extension)
  * (u) : movu → treat the 16 bit immediate as an unsigned number
  * (h) : movh → left shift the 16 bit immediate by 16 positions

Mechanism

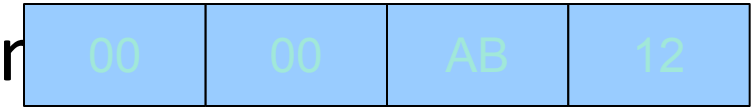
* The processor internally converts a 16 bit immediate to a 32 bit number
* It uses this 32 bit number for all the computations
* Valid only for arithmetic/logical insts
* We can control the generation of this 32 bit number
  * sign extension (default)
  * treat the 16 bit number as unsigned (u suffix)

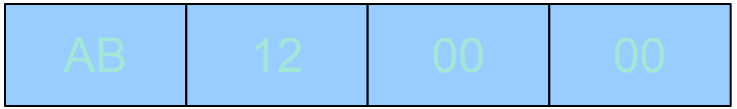
More about Modifiers

* default : mov r1, 0xAB 12



* unsigned : r





high: movh r1, 0xAB 12

Examples

* Move : 0x FF FF A3 2B in r0

```
mov r0, 0xA32B
```

* Move : 0x 00 0

```
movu r0, 0xA32B
```

```
movh r0, 0xA32B
```

* Move : 0x A3 2B 00 00 in r0

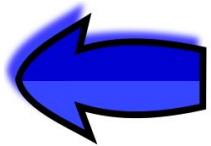
Example

* Set r0 ← 0x 12 AB A9 2D

```
movh r0, 0x 12 AB
addu r0, 0x A9 2D
```

Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA
* Functions and Stacks
* SimpleRisc Encoding

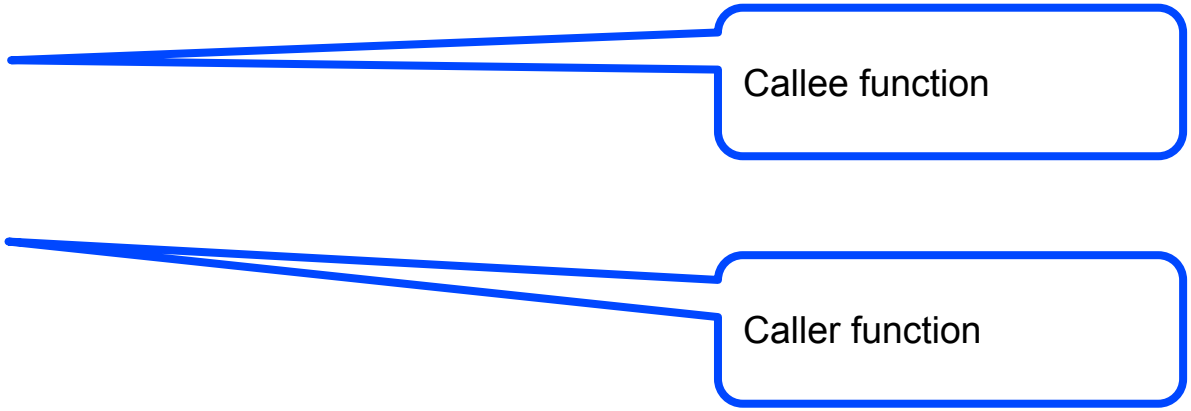


Functions

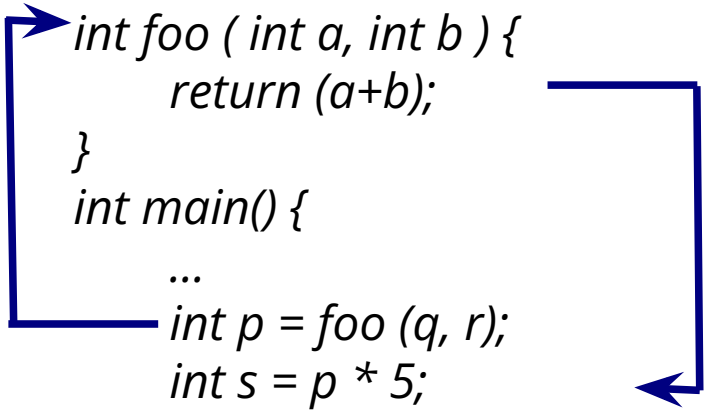
- Instruction sequence that is used frequently
- Typically invoked in more than one context
  - Can even invoke itself
- Notion of input arguments
- Notion of return values
- Notion of state

Illustrative Example

```
int foo ( int a, int b ) {
        return (a+b);
}
int main() {
        ...
        int p = foo (q, r);
        int s = p * 5;
        ...
        int w = foo (x, y);
        int z = w - 3;
        ...
}
```



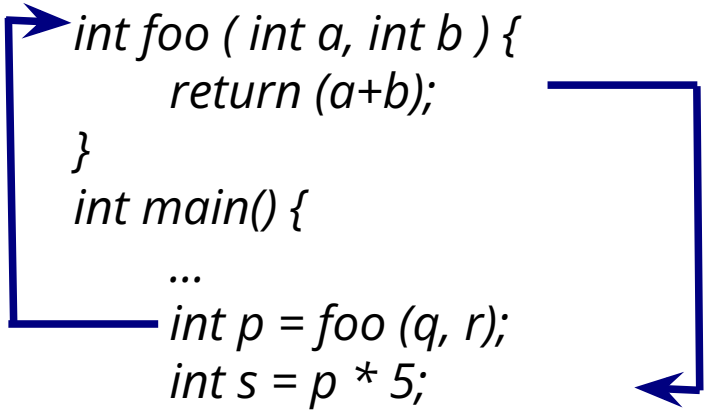
Illustrative Example



```
int foo ( int a, int b ) {
        return (a+b);
}
int main() {
        ...
        int p = foo (q, r);
        int s = p * 5;
        ...
        int w = foo (x, y);
        int z = w - 3;
        ...
}
```

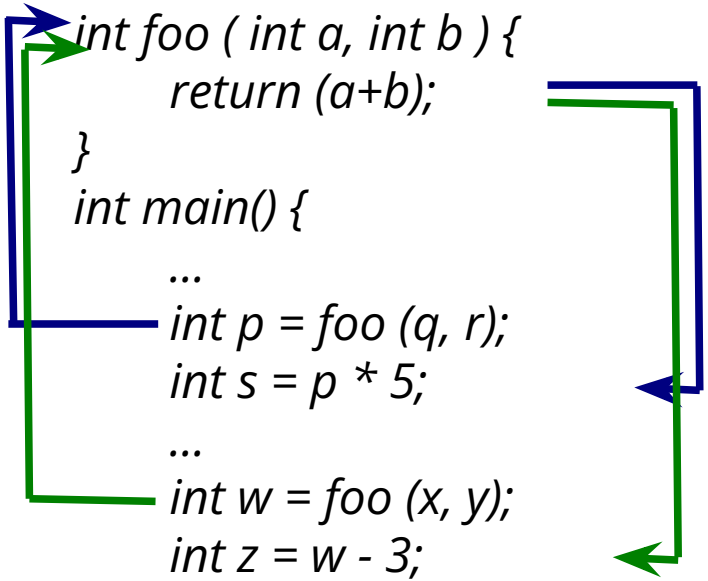
Illustrative Example

```
int foo ( int a, int b ) {
        return (a+b);
}
int main() {
        ...
        int p = foo (q, r);
        int s = p * 5;
        ...
        int w = foo (x, y);
        int z = w - 3;
        ...
}
```



Desired Control Flow

Illustrative Example

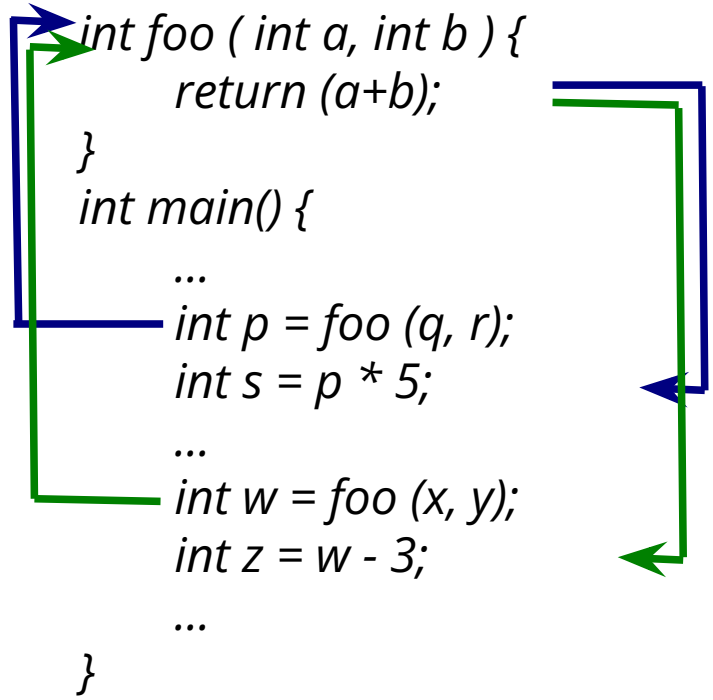


Two New Instructions in the ISA

1. call .foo
    a. ra ← PC + 4
    b. PC ← address(.foo)
2. ret
    a. PC ← ra

Remember:
“ra” or the Return Address register is the same as r15

Illustrative Example: Work out the Control Flow



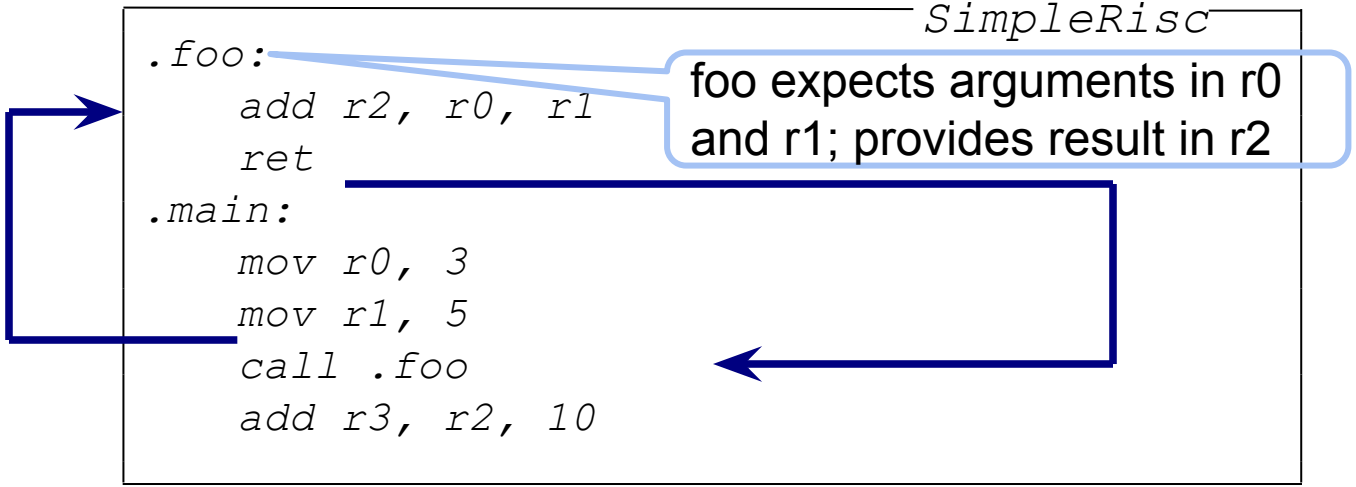
```
int foo ( int a, int b ) {
        return (a+b);
}
int main() {
        ...
        int p = foo (q, r);
        int s = p * 5;
        ...
        int w = foo (x, y);
        int z = w - 3;
        ...
}
```

```
.foo:
0x1234  ...
0x1238  ret
...
.main:
0x2000  ...
...
0x2040  call .foo
0x2044  ...
...
0x2100  call .foo
0x2104  ...
...
```

PC	ra
0x2040	

How do we pass arguments/ return values

* Solution : use registers



Problems with this Mechanism

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than 16 arguments

Illustrative Example

```
int factorial ( int n ) {
        if ( n == 1 )
                return n;
        else
                return n * factorial ( n - 1 );
}
int main() {
        factorial ( 3 );
}
```

```
.factorial:
0x1000      cmp r2, 1
0x1004      beq .done
0x1008      mov r3, r2
0x100c      sub r2, r2, 1
0x1010      call .factorial
0x1014      mul r1, r1, r3
0x1018      ret
.done:
0x101c      mov r1, 1
0x1020      ret
...
.main:
0x2000      mov r2, 3
0x2004      call .factorial
```

factorial expects argument in r2; provides result in r1

Illustrative Example

```
int factorial ( int n ) {
        if ( n == 1 )
                return n;
        else
                return n * factorial ( n - 1 );
}
int main() {
        factorial ( 3 );
}
```

Value of r3, which constituted the "state" of the particular invocation/ call of the function, is getting lost.

```
.factorial:
0x1000      cmp r2, 1
0x1004      beq .done
0x1008      mov r3, r2
0x100c      sub r2, r2, 1
0x1010      call .factorial
0x1014      mul r1, r1, r3
0x1018      ret
.done:
0x101c      mov r1, 1
0x1020      ret
...
.main:
0x2000      mov r2, 3
0x2004      call .factorial
```

factorial expects argument in r2; provides result in r1

Problems with this Mechanism

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than 16 arguments
* Overwrite Problem
  * What if a function calls itself ? (recursive call)
  * The callee can overwrite the registers of the caller

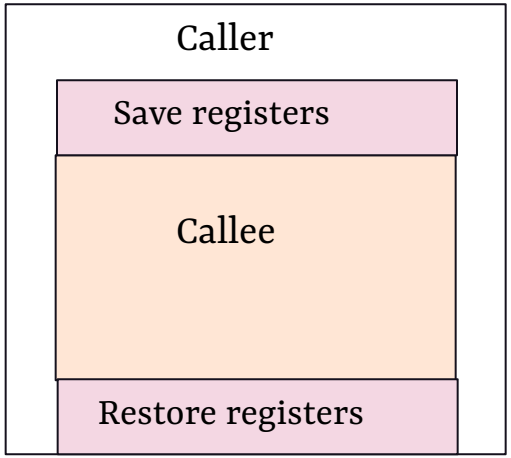
Problems with this Mechanism

* Space Problem
  * We have a limited number of registers
  * We cannot pass more than 16 arguments
  * Solution : Use memory also
* Overwrite Problem
  * What if a function calls itself ? (recursive call)
  * The callee can overwrite the registers of the caller
  * Solution : Spilling

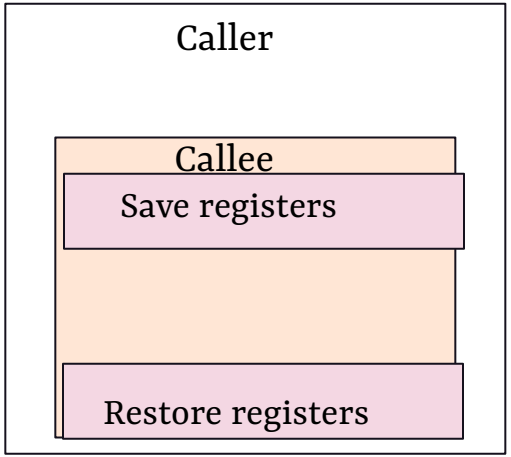
Register Spilling

* The notion of spilling
  * The caller can save the set of registers its needs
  * Call the function
  * And then restore the set of registers after the function returns
  * Known as the caller saved scheme
* callee saved scheme
  * The callee saves, the registers, and later restores them

Spilling



(a) Caller saved



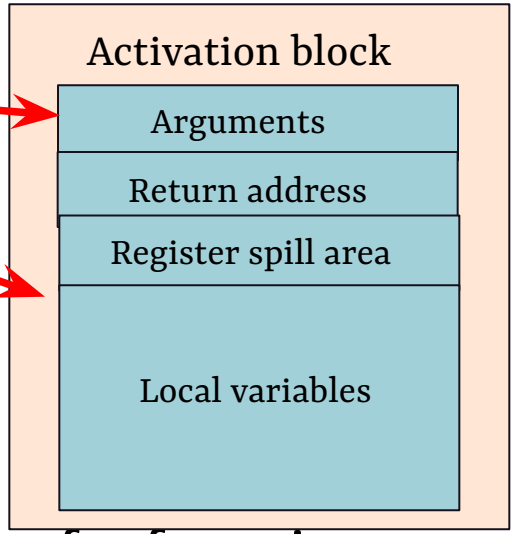
(b) Callee saved

Problems with our Approach

* Using memory, and spilling
  solves both the space problem and overwrite problem
* However, there needs to be :
  * a strict agreement between the caller and the callee regarding the set of memory locations that need to be used
  * Secondly, after a function has finished execution, all the space that it uses needs to be reclaimed

Activation Block

```
int foo(int arg1) {
    int a, b, c;
    a = 3; b = 4;
    c = a + b + arg1;
    return c;
}
```



* Activation block → memory map of a function

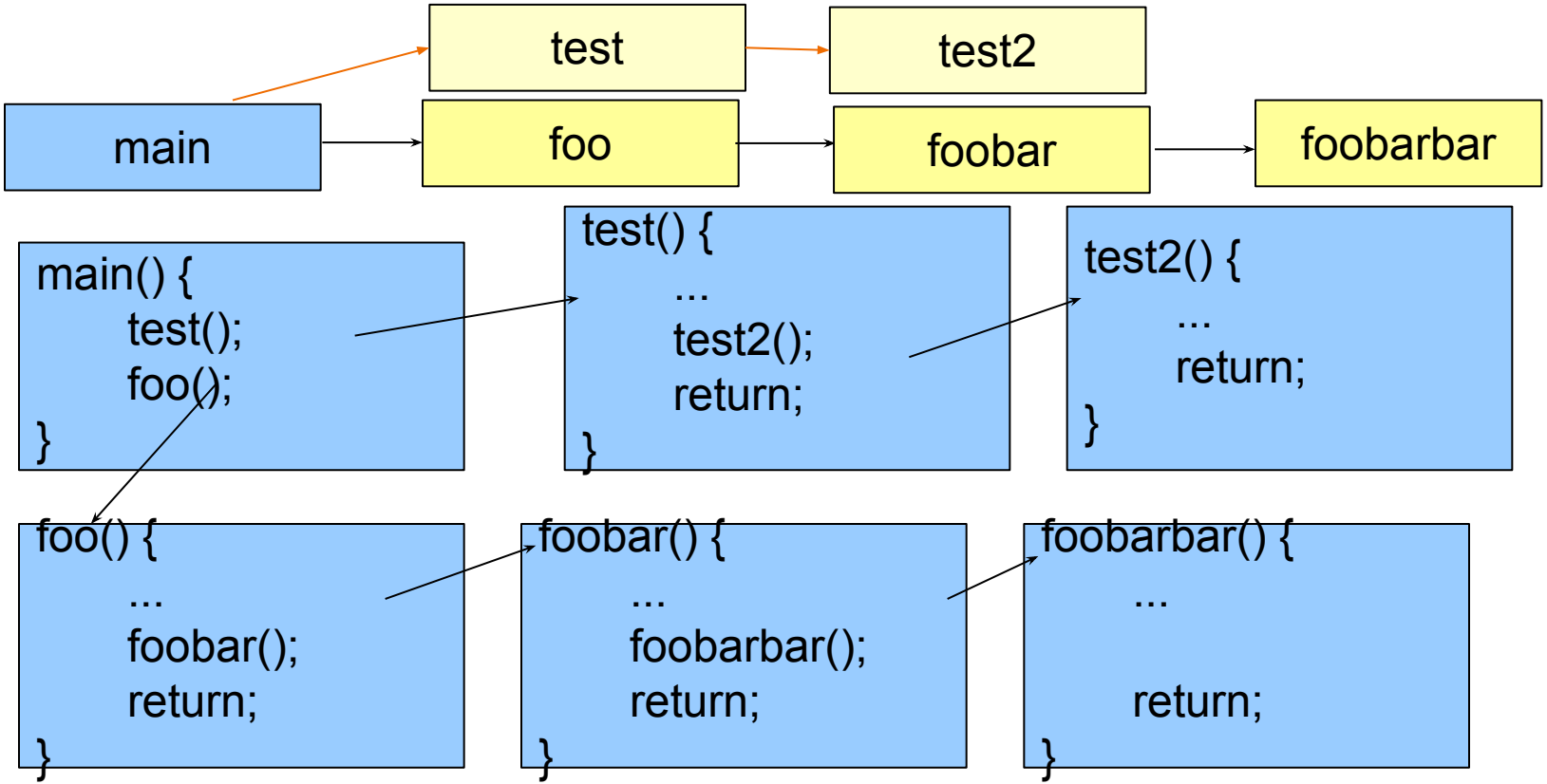
  arguments, register spill area, local vars

Organising Activation Blocks

* All the information of an executing function is stored in its activation block
* These blocks need to be dynamically created and destroyed – millions of times
* What is the correct way of managing them, and ensuring their fast creation and deletion ?

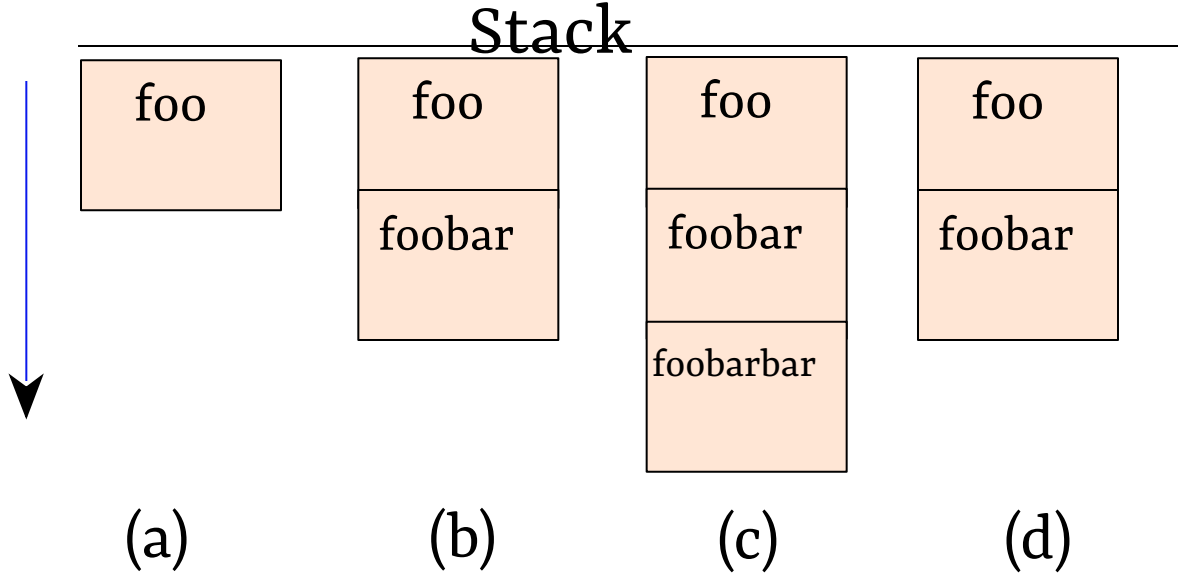
Is there a pattern ?

Pattern of Function Calls



Pattern of Function Calls

* Last in First Out
* Use a stack to store activation blocks



Working with the Stack

* Allocate a part of the memory to save the stack
* Traditionally stacks are downward growing.
  * The first activation block starts at the highest address
  * Subsequent activation blocks are allocated lower addresses
* The stack pointer register (sp (14)) points to the beginning of an activation block
* Allocating an activation block :
  * sp ⟵ sp - <constant>

What has the Stack Solved ?

* Space problem
  * Pass as many parameters as required in the activation block
* Overwrite problem
  * Solved by activation blocks
* Management of activation blocks
  * Solved by the notion of the stack

The stack needs to primarily be managed in software

call and ret instructions

call .foo	*ra* ← *PC* + 4 ; *PC* ← *address*(*.foo*);
ret	*PC* ← *ra*

* ra (or r15) ← return address register
* call instruction
  * Puts *pc + 4* in ra, and jumps to the function
* ret instruction

How to use activation blocks ?

* Assume caller saved spilling
* Before calling a function : spill the registers
    - This includes the *ra* register
* Allocate the activation block of the callee
* Write the arguments to the activation block of the callee, if they do not fit in registers
* Call the function

Using Activation Blocks - II

* In the called function
  * Read the arguments and transfer to registers (if required)
  * Save the return address if the called function can call other functions
  * Allocate space for local variables
  * Execute the function
* Once the function ends
  * Restore the value of the return address register (if required)
  * Write the return values to registers, or the activation block of the caller

Using Activation Blocks - III

* Once the function ends (contd …)
  * Call the ret instruction
  * and return to the caller
* The caller :
  * Retrieve the return values from the registers of from its activation block
  * Restore the spilled registers

Recursive Factorial Program

C

```
int factorial(int num) {
    if (num <= 1) return 1;
    return num * factorial(num - 1);
}
void main() {
    int result = factorial(10);
}
```

Factorial in SimpleRisc

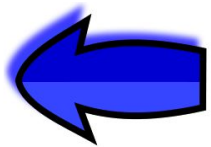
```
.factorial:
0    cmp r0, 1            /* compare (1,num) */
4    beq .return
8    bgt .continue
12   b .return
.continue:
16   sub sp, sp, 8        /* create space on the stack */
20   st r0, [sp]          /* push r0 on the stack */
24   st ra, 4[sp]         /* push the return address register */
28   sub r0, r0, 1        /* num = num - 1 */
32   call .factorial      /* result will be in r1 */
36   ld r0, [sp]          /* pop r0 from the stack */
40   ld ra, 4[sp]         /* restore the return address */
44   mul r1, r0, r1       /* factorial(n) = n * factorial(n-1) */
48   add sp, sp, 8        /* delete the activation block */
52   ret
.return:
56   mov r1, 1
60   ret
.main:
64   mov r0, 10
68   call .factorial      /* result in r1 */
```

nop instruction

* nop ⟶ does nothing
* Example : nop

Outline

* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA
* Functions and Stacks
* SimpleRisc Encoding



Encoding Instructions

* Encode the SimpleRisc ISA using 32 bits.
* We have 21 instructions. Let us allot each instruction an unique code (opcode)

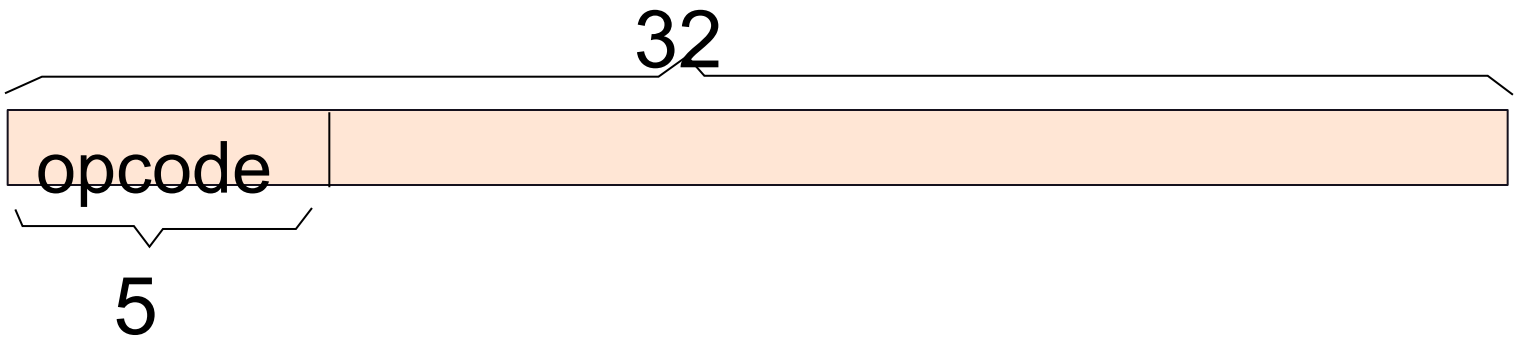
Instruction	Code	Instruction	Code	Instruction	Code
add	0000	not	0100	beq	1000
sub	0000	mov	0100	bgt	1000
mul	0001	lsl	0101		1001
div	0001	lsr	0101	call	1001
mod	0010	asr	0110	ret	1010
cmp	0010	nop	0110		
and	0011	ld	0111		
or	0011		0111		

Basic Instruction Format

Inst.	Code	Form	Inst.	Code	Form
add	0000	add rd, rs1, (rs2/imm)	lsl	0101	lsl rd, rs1, (rs2/imm)
sub	0000	sub rd, rs1, (rs2/imm)	lsr	0101	lsr rd, rs1, (rs2/imm)
mul	0001	mul rd, rs1, (rs2/imm)	asr	0110	asr rd, rs1, (rs2/imm)
div	0001	div rd, rs1, (rs2/imm)	nop	0110	nop
mod	0010	mod rd, rs1, (rs2/imm)		0111	ld rd, imm[rs1]
cmp	0010	cmp rs1, (rs2/imm)		0111	st rd, imm[rs1]
and	0011	and rd, rs1, (rs2/imm)	beq	1000	beq offset
or	0011	or rd, rs1, (rs2/imm)	bgt	1000	bgt offset
not	0100	not rd, (rs2/imm)		1001	b offset
mov	0100	mov rd, (rs2/imm)	call	1001	call offset
			ret	1010	ret

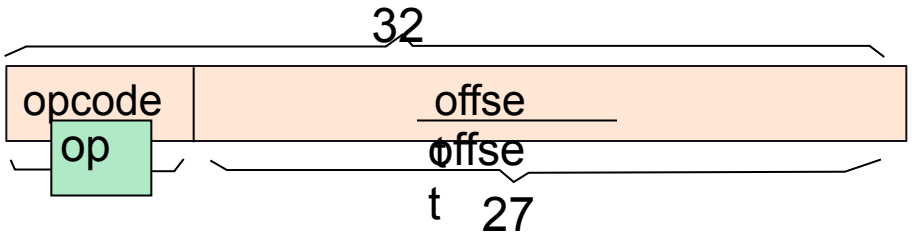
0-Address Instructions

* nop and ret instructions



5

1-Address Instructions

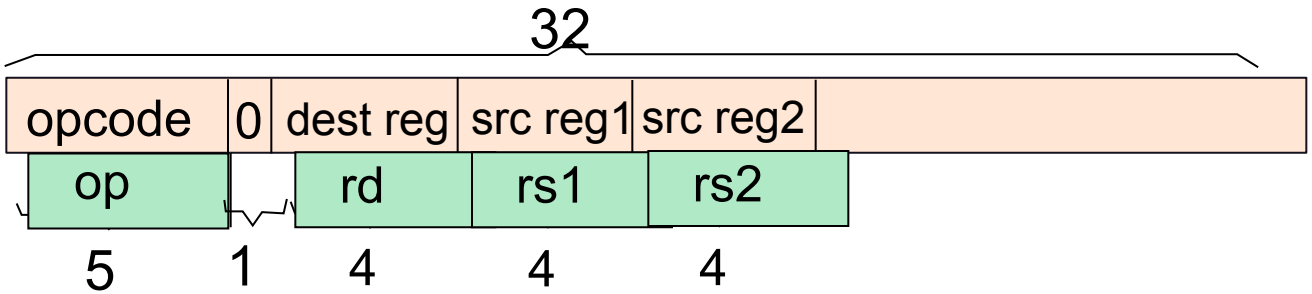


* Instructions – call, b, beq, bgt
* Use the branch format
* Fields :
  * 5 bit opcode
  * 27 bit offset (PC relative addressing)

3-Address Instructions

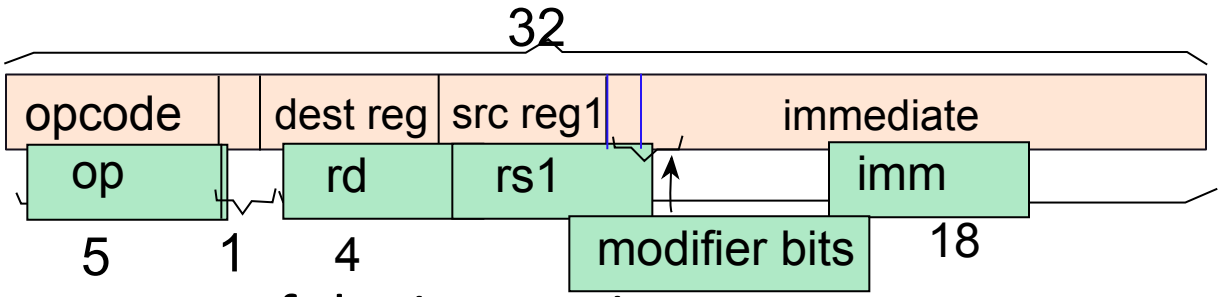
* Instructions – add, sub, mul, div, mod, and, or, lsl, lsr, asr
* Generic 3 address instruction
  * <opcode> rd, rs1, <rs2/imm>
* Let us use the I bit to specify if the second operand is an immediate or a register.
  * I = 0 ⟶ second operand is a register
  * I = 1 ⟶ second operand is an immediate

Register Format



* opcode → type of the instruction
* I bit → 0 (second operand is a register)
* dest reg → rd
* source register 1 → rs1
* source register 2 → rs2

Immediate Format



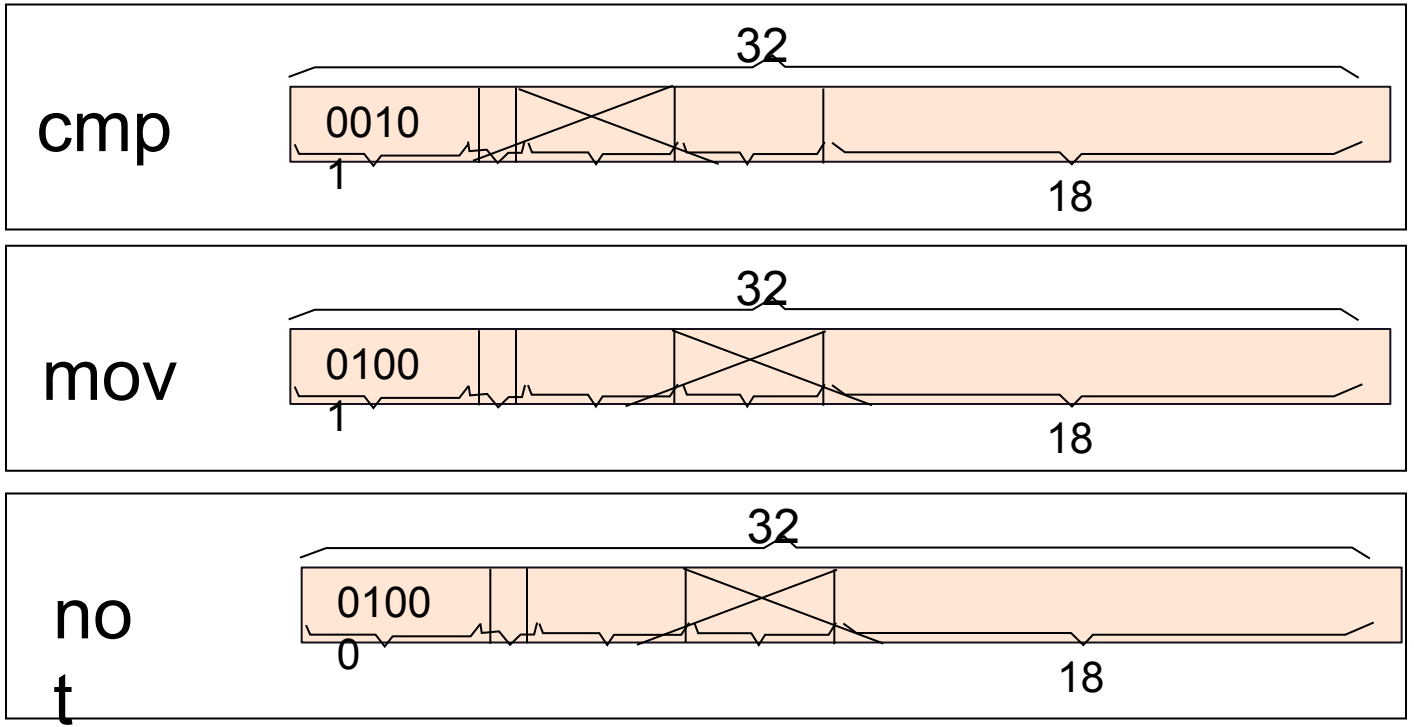
* opcode → type of the instruction
* I bit → 1 (second operand is an immediate)
* dest reg → rd
* source register 1 → rs1
* Immediate → imm
* modifier bits → 00 (default), 01 (u), 10 (h)

2 Address Instructions

* cmp, not, and mov
* Use the 3 address : immediate or register formats
* Do not use one of the fields

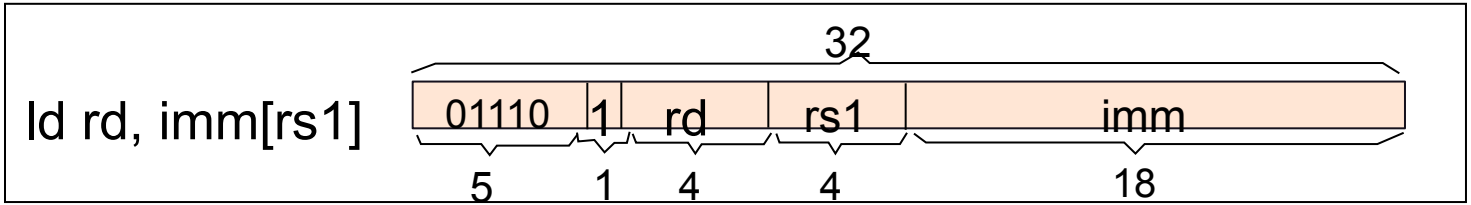
$s2//imm

cmp, not, and mov



Load and Store Instructions

* ld rd, imm[rs1]
* rs1 → base register
* Use the immediate format.



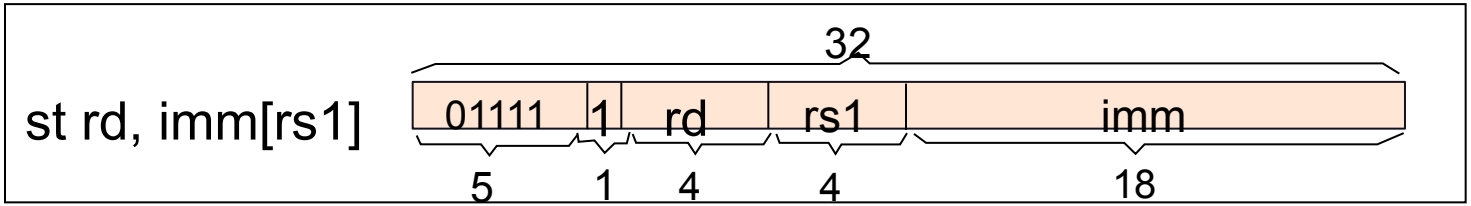
Store Instruction

* Strange case of the store inst.
* st reg1, imm[reg2]
* has two register sources, no register destination, 1 immediate
* Cannot fit [illegible] immediate format, because the second operand can be eit[illegible] a register OR an immediate (not both)

* Should we define a new form[illegible]e instructions ?

Store Instruction

* Let us make an exception and use the immediate format.
* We use the rd field to save one of the source registers
* st rd, imm[rs1]



Summary of Instruction Formats

Form at	(28-3	(1-27)			
	6)8-3	(27)	(23-2	1(19-22)	2(15-18)
	6)8-3	(27)	6)3-2	1(19-22)	(1-18)

op → opcode, *offset*2) → branch offset, *I* → immediate bit, *rd* → destination register
*rs*1 → source register 1, *rs*2 → source register 2, *imm* → immediate operand

* branch format → nop, ret, call, b, beq, bgt
* register format → ALU instructions
* immediate format → ALU, ld/st instructions