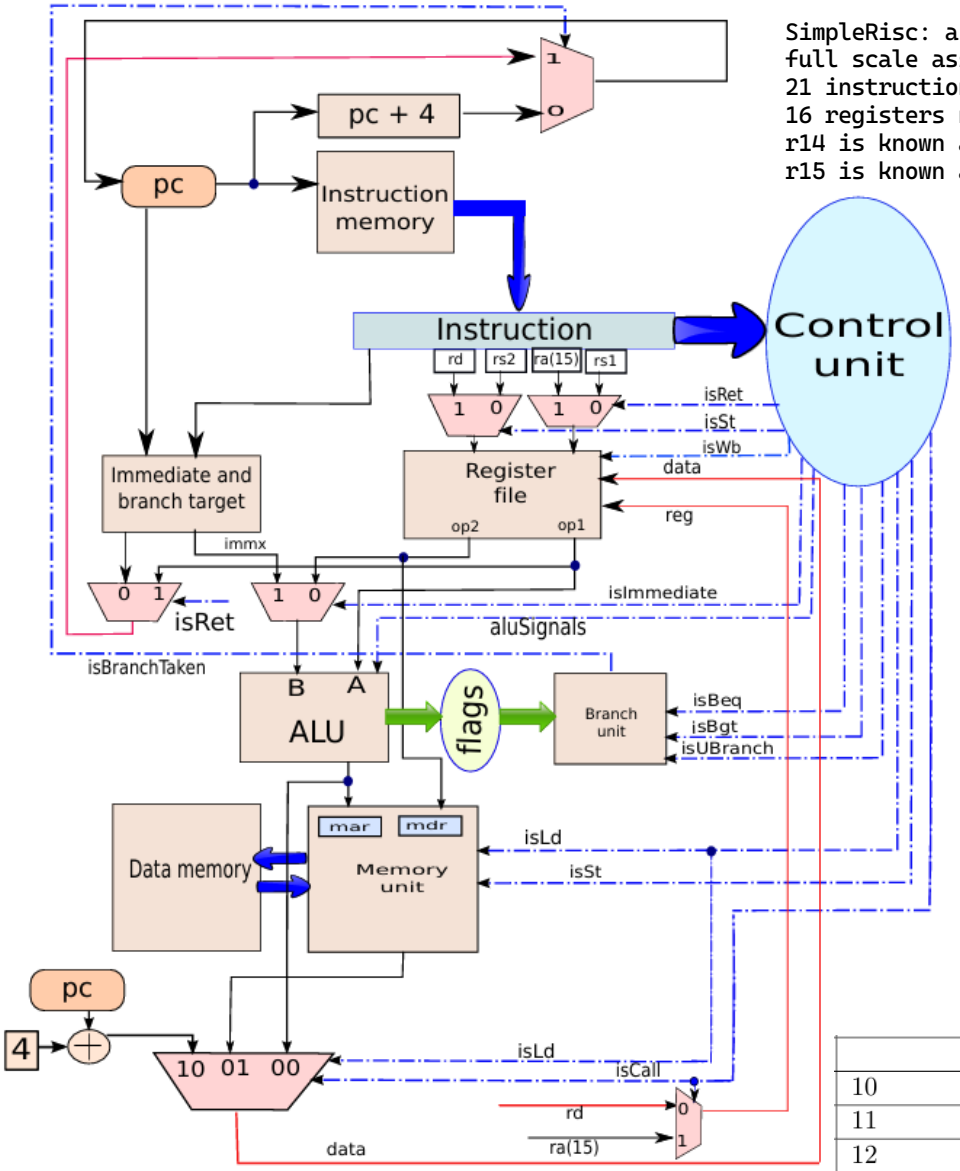


SimpleRisc: a simple, generic, complete and concise ISA that captures most of the features of full scale assembly languages. It has:
21 instructions
16 registers numbered r0 r15. first 14 registers are general purpose registers.
r14 is known as the stack pointer register (sp).
r15 is known as the return address register (ra).

Instruction	Code	Instruction	Code	Instruction	Code
add	00000	not	01000	beq	10000
sub	00001	mov	01001	bgt	10001
mul	00010	lsl	01010	b	10010
div	00011	lsr	01011	call	10011
mod	00100	asr	01100	ret	10100
cmp	00101	nop	01101		
and	00110	ld	01110		
or	00111	st	01111		

Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs1* (19-22)	*rs2* (15-18)
immediate	*op* (28-32)	*I* (27)	*rd* (23-26)	*rs1* (19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
rs1 → source register 1, *rs2* → source register 2, *imm* → immediate operand					

Serial No.	Signal	Condition
1	$isSt$	$\overline{op_5}.op_4.op_3.op_2.op_1$
2	$isLd$	$\overline{op_5}op_4.op_3.op_2.\overline{op_1}$
3	$isBeq$	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.\overline{op_1}$
4	$isBgt$	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.op_1$
5	$isRet$	$op_5.\overline{op_4}.op_3.\overline{op_2}.\overline{op_1}$
6	$isImmediate$	I
7	$isWb$	$\sim(op_5 + \overline{op_5}.op_3.op_1.(op_4 + \overline{op_2})) + op_5.\overline{op_4}.\overline{op_3}.op_2.op_1$
8	$isUbranch$	$op_5.\overline{op_4}.(\overline{op_3}.op_2 + op_3.\overline{op_2}.\overline{op_1})$
9	$isCall$	$op_5.\overline{op_4}.\overline{op_3}.op_2.op_1$

aluSignals		
10	$isAdd$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.\overline{op_2}.\overline{op_1} + \overline{op_5}.op_4.op_3.op_2$
11	$isSub$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.\overline{op_2}.op_1$
12	$isCmp$	$\overline{op_5}.\overline{op_4}.op_3.\overline{op_2}.op_1$
13	$isMul$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.op_2.\overline{op_1}$
14	$isDiv$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.op_2.op_1$
15	$isMod$	$\overline{op_5}.\overline{op_4}.op_3.\overline{op_2}.\overline{op_1}$
16	$isLsl$	$\overline{op_5}.op_4.\overline{op_3}.op_2.\overline{op_1}$
17	$isLsr$	$\overline{op_5}.op_4.\overline{op_3}.op_2.op_1$
18	$isAsr$	$\overline{op_5}.op_4.op_3.\overline{op_2}.\overline{op_1}$
19	$isOr$	$\overline{op_5}.\overline{op_4}.op_3.op_2.op_1$
20	$isAnd$	$\overline{op_5}.\overline{op_4}.op_3.op_2.\overline{op_1}$
21	$isNot$	$\overline{op_5}.op_4.\overline{op_3}.\overline{op_2}.\overline{op_1}$
22	$isMov$	$\overline{op_5}.op_4.\overline{op_3}.\overline{op_2}.op_1$

Instruction	Value of $isBranchTaken$
non-branch instruction	0
call	1
ret	1
b	1
beq	branch taken – 1 branch not taken – 0
bgt	branch taken – 1 branch not taken – 0

Semantics	Example	Explanation
call *label*	call .foo	$ra \leftarrow PC + 4$; $PC \leftarrow address(.foo)$;
ret	ret	$PC \leftarrow ra$

isWb	Instructions: *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *not*, *mov*, *ld*, *lsl*, *lsr*, *asr*, *call*
isUBranch	Instructions: *b*, *call*, *ret*
isAdd	Instructions: *add*, *ld*, *st*

Semantics	Example	Explanation
b *label*	b .foo	branch to .foo

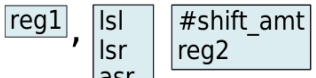
Semantics	Example	Explanation
add *reg*, *reg*, (*reg*/*imm*)	add r1, r2, r3	$r1 \leftarrow r2 + r3$
	add r1, r2, 10	$r1 \leftarrow r2 + 10$
sub *reg*, *reg*, (*reg*/*imm*)	sub r1, r2, r3	$r1 \leftarrow r2 - r3$
mul *reg*, *reg*, (*reg*/*imm*)	mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
div *reg*, *reg*,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

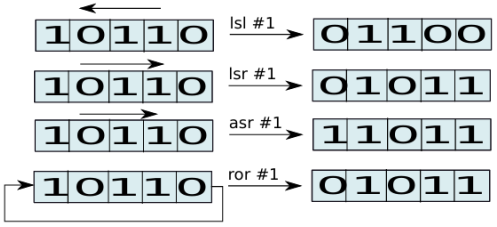
Semantics	Example	Explanation
beq *label*	beq .foo	branch to .foo if $flags.E = 1$
bgt *label*	bgt .foo	branch to .foo if $flags.GT = 1$

Semantics	Example	Explanation
ld *reg*, *imm*[*reg*]	ld r1, 12[r2]	$r1 \leftarrow [r2 + 12]$
st *reg*, *imm*[*reg*]	st r1, 12[r2]	$[r2 + 12] \leftarrow r1$

Semantics	Example	Explanation
and *reg*, *reg*, (*reg*/*imm*)	and r1, r2, r3	$r1 \leftarrow r2 \wedge r3$
or *reg*, *reg*, (*reg*/*imm*)	or r1, r2, r3	$r1 \leftarrow r2 \vee r3$
not *reg*, (*reg*/*imm*)	not r1, r2	$r1 \leftarrow \sim r2$
$\wedge$ bitwise AND, $\vee$ bitwise OR, $\sim$ logical complement		

Semantics	Example	Explanation
lsl *reg*, *reg*, (*reg*/*imm*)	lsl r3, r1, r2	$r3 \leftarrow r1 \ll r2$ (shift left)
	lsl r3, r1, 4	$r3 \leftarrow r1 \ll 4$ (shift left)
lsr *reg*, *reg*, (*reg*/*imm*)	lsr r3, r1, r2	$r3 \leftarrow r1 \ggg r2$ (shift right logical)
	lsr r3, r1, 4	$r3 \leftarrow r1 \ggg 4$ (shift right logical)
asr *reg*, *reg*, (*reg*/*imm*)	asr r3, r1, r2	$r3 \leftarrow r1 \gg r2$ (arithmetic shift right)
	asr r3, r1, 4	$r3 \leftarrow r1 \gg 4$ (arithmetic shift right)





Semantics	Example	Explanation
mov *reg*, (*reg*/*imm*)	mov r1, r2	$r1 \leftarrow r2$
	mov r1, 3	$r1 \leftarrow 3$

Modifiers

how to load the constant 0xFB12CDEF ? this is a 32 bit value
but our instructions only support supplying a 16 bit value

```
/* load the upper two bytes */
mov r0, 0xFB12
lsl r0, r0, 16

/* load the lower two bytes with 0x CD EF */
mov r1, 0xCDEF
lsl r1, r1, 16
lsr r1, r1, 16 /* top 16 bits are zeros */

/* load all the four bytes */
add r0, r0, r1
```

with 'u' and 'h' modifiers:

```
movh r0, 0xFB12      /* r0 = 0xFB 12 00 00 */
addu r0, r0, 0xCDEF  /* r0 = r0 + 0x00 00 CD EF */
```

Default (00): performs sign extension on the 16-bit immediate,
fills the upper 16 bits with 1's when converting to 32 bits.
Example: -2 (represented as 0xFFFE) would result in 0xFFFFFFFE in the 32-bit register.

'u' (01): considers 16-bit immediate as an unsigned number,
filling the top 16 bits with zeros.
Example: movu r0, 0xFEAB would load 0x0000FEAB into register r0

'h' (10): load the 16-bit immediate into the upper half of a register,
effectively shifting it 16 positions to the left.
Example: movh r0, 0xFEAB would load 0xFEAB0000 into r0

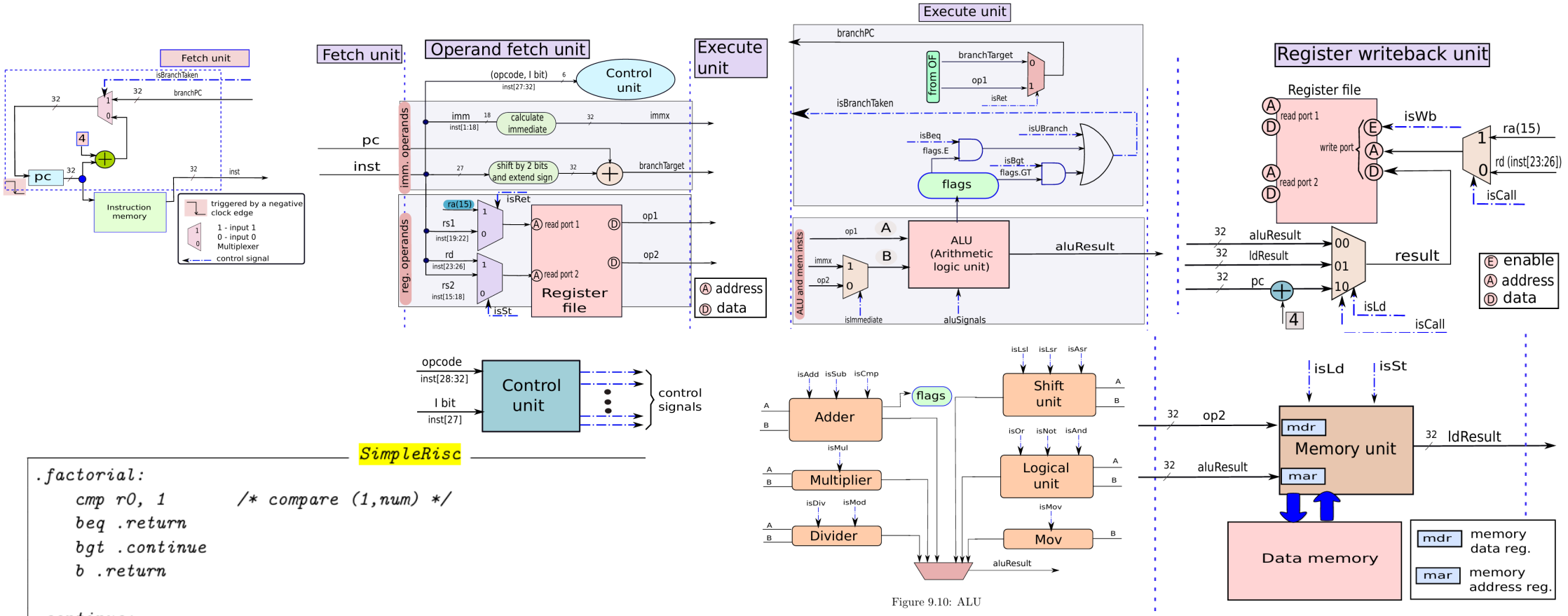


Figure 9.10: ALU

SimpleRisc

```
.factorial:
    cmp r0, 1        /* compare (1,num) */
    beq .return
    bgt .continue
    b .return

.continue:
    sub sp, sp, 8    /* create space on the stack */
    st r0, [sp]      /* push r0 on the stack */
    st ra, 4[sp]     /* push the return address register */
    sub r0, r0, 1    /* num = num - 1 */
    call .factorial /* result will be in r1 */
    ld r0, [sp]      /* pop r0 from the stack */
    ld ra, 4[sp]     /* restore the return address */
    mul r1, r0, r1  /* factorial(n) = n * factorial(n-1) */
    add sp, sp, 8    /* delete the activation block */
    ret
.return:
    mov r1, 1
    ret

.main:
    mov r0, 10
    call .factorial
```

C

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