The SimProc instruction set

This page tells you what every SimProc opcode does, how it is encoded, and how many cycles it takes. It's meant to be kept open in a tab while you work. If you want the machine explained rather than tabulated, read SimProc in one page instead.

You can check everything here against two files that ship with the course: the assembler, `labs/common/simproc_asm.py`, and the model, `labs/common/golden.py`. Both run at a plain Python prompt, without a simulator.

Encoding

Every instruction is one byte, and three different field layouts share those eight bits:

```
register form     A A B B o o o o      rA = instr[7:6]   rB = instr[5:4]
                  7 6 5 4 3 2 1 0      op = instr[3:0]

branch form       d d d d o o o o      d  = instr[7:4], 4-bit SIGNED
                  7 6 5 4 3 2 1 0      op = instr[3:0]

SHIFT             A A D S S 0 1 1      D  = instr[5]: 1 = left, 0 = right
                  7 6 5 4 3 2 1 0      SS = instr[4:3], the amount, 0..3

ORI               i i i i i 1 1 1      i  = instr[7:3], a 5-bit immediate
                  7 6 5 4 3 2 1 0
```

`SHIFT` and `ORI` are selected by their low three bits, while every other instruction is selected by the full low nibble.

The table

`rA` is both a source and the destination. The flags are `N`, which is bit 7 of the result, and `Z`, which is set when the whole result was zero. A dash in the flags column means the instruction leaves both flags exactly as they were.

op	byte	mnemonic	what it does	flags	cycles	assembler
`0000`	`0x?0`	`LOAD rA, rB`	`rA <- mem[rB]`; rB is the pointer	—	4	`LOAD(1, 2)`
`0001`	`0x?1`	`JMP d`	`PC <- PC+1+d`, always	—	3	`JMP(-3)`
`0010`	`0x?2`	`STORE rA, rB`	`mem[rB] <- rA`; same order as `LOAD`	—	3	`STORE(1, 3)`
`0011`	`0x?3`	`SHL rA, n` / `SHR rA, n`	`rA <- rA << n` or `rA >> n`, `n` in 0..3	`N Z`	4	`SHL(0, 3)`
`0100`	`0x?4`	`ADD rA, rB`	`rA <- rA + rB`, wraps at 8 bits	`N Z`	4	`ADD(1, 0)`
`0101`	`0x?5`	`BPZ d`	if `N == 0`: `PC <- PC+1+d`	—	3	`BPZ(2)`

op	byte	mnemonic	what it does	flags	cycles	assembler
0110	0x?6	SUB rA, rB	rA <- rA - rB, wraps at 8 bits	N Z	4	SUB(1, 2)
0111	0x?7	ORI imm5	r0 <- r0 | imm5; the destination is always r0	N Z	5	ORI(5)
1000	0x?8	NAND rA, rB	rA <- ~(rA & rB)	N Z	4	NAND(1, 1)
1001	0x?9	BNZ d	if Z == 0: PC <- PC+1+d	—	3	BNZ(-3)
1010	0x?A	BZ d	if Z == 1: PC <- PC+1+d	—	3	BZ(1)
1011	0x?B	*(SHIFT again)*	the second encoding of SHIFT	N Z	4	
1100	0x?C	*(unimplemented)*	PC+1 and does nothing else	—	4	UNIMP()
1101	0x?D	*(unimplemented)*	PC+1 and does nothing else	—	4	
1110	0x?E	*(unimplemented)*	PC+1 and does nothing else	—	4	
1111	0x?F	*(ORI again)*	the second encoding of ORI	N Z	5	

The ? stands for whatever the register or displacement fields happen to hold. The byte 0x44 is ADD r1, r0 and the byte 0x04 is ADD r0, r0, but both are ADD, because the low nibble is 4 in each.

All 256 bytes

There are no illegal encodings, because every byte decodes to something, and no two opcodes collide. We checked this by single-stepping all 256 instruction bytes and classifying each one by its cycle count, done, PC delta, register write and memory write:

	encodings	which
the nine 4-bit opcodes	16 each, 144 total	0x?0 0x?1 0x?2 0x?4 0x?5 0x?6 0x?8 0x?9 0x?A
SHIFT	32	0x?3 and 0x?B
ORI	32	0x?7 and 0x?F

	encodings	which
unimplemented	48	`0x?C` `0x?D` `0x?E`
	256	

Because `SHIFT` and `ORI` are chosen by only three bits, each of them owns two low nibbles. Nothing else is assigned to `0x?B` or `0x?F`, so the aliasing does no harm. It does mean, though, that the byte `0xF1` is a `JMP` while `0xFF` is an `ORI`, and the two differ in one bit that most people don't look at.

Disassembling a byte from the command line

The assembler also works as a disassembler. When a test fails and a log line shows you a hex byte, this is the quickest way to find out which instruction it was:

```
cd labs/common
python3 simproc_asm.py 0x2F 0x44 0xD9
```

```
0x2F   ORI   5
0x44   ADD   r1, r0
0xD9   BNZ   -3
```

The script imports nothing and needs no simulator, and it will decode as many bytes as you give it.

Here are some bytes worth learning to recognise on sight:

byte		
`0x0C`	`(unimplemented)`	what the harness fills unwritten RAM with
`0xF1`	`JMP -1`	the parking loop, which is a jump to itself
`0x06`	`SUB r0, r0`	the idiom for "clear `r0`"
`0x2F`	`ORI 5`	
`0x3B`	`SHL r0, 3`	
`0x8B`	`SHR r2, 1`	note the low nibble, `B`, which is `SHIFT`'s second encoding

Branch displacements

The PC is incremented during `CYCLE_1`, *before* the branch computes its target in `CYCLE_3`, so the displacement is measured from the instruction **after** the branch:

$$\text{target} = (\text{address of the branch}) + 1 + d$$

`JMP(0)`	falls through to the next instruction
`JMP(1)`	skips one instruction
`JMP(-1)`	an infinite loop on itself, which is the parking idiom
range	`-8` to `+7`, and nothing outside it

The field is four bits wide and **signed**. If you branch to a label further away than that, the assembler stops with an error rather than quietly wrapping the displacement:

```
ValueError: line 12: 'BNZ loop': 'loop' is -11 instructions away and a SimProc
branch only reaches -8..+7. Move the label closer, or JMP to a JMP.
```

Flags

set `N` and `Z`	leave them alone
`ADD SUB NAND SHL SHR ORI`	`LOAD STORE JMP BZ BNZ BPZ`

`N` is bit 7 of the result, and `Z` means the whole 8-bit result was zero. There is no carry flag and no overflow flag, so arithmetic wraps without telling you.

The branches leave the flags alone by specification, not by accident, and that is what lets you compare once and then branch on the result twice:

```
        SUB   r1, r1        # r1 = 0  ->  Z = 1
        BZ    taken         # taken
        ORI   16            # skipped
taken:  BNZ   wrong         # must NOT be taken: Z is still 1
```

Idioms

This instruction set lacks several things you're used to, and each gap has a standard workaround. Think of these as the only way to write each operation on SimProc rather than as tricks.

you want	you write	why
clear `rX`	`SUB rX, rX`	there is no `MOV` and no `CLR`
`NOT rX`	`NAND rX, rX`	there is no `NOT`
compare `rA` against `rB`	`SUB rA, rB`	there is no `CMP`, and the compare destroys `rA`
stop the program	`JMP -1`	there is no `HALT` instruction
a constant in `r0`	`SUB r0, r0`, then `ORI`	`ORI` only *sets* bits
a constant larger than 31	`ORI` the high bits, `SHL`, `ORI` the low bits	the immediate is 5 bits
a constant in `r1`/`r2`/`r3`	build it in `r0`, then `SUB rX, rX ; ADD rX, r0`	`ORI` only writes `r0`

> Every row in that table is also an entry in the bug catalogue. `SUB rX, rX` is symmetric, so it gives the right answer even on a core whose subtract is backwards, and `NAND rX, rX` is the one operand pattern for which `NAND` and `NOR` agree. The idiom a programmer reaches for first is the one that hides the bug. Where bugs hide covers this at length.

Building a constant by hand is easy to get wrong, so the assembler has a helper that does it for you:

```
>>> from simproc_asm import set_reg, listing
>>> print(listing(set_reg(1, 0xB7)))
  00: 06   SUB   r0, r0
  01: B7   ORI   22
  02: 3B   SHL   r0, 3
  03: 3F   ORI   7
  04: 56   SUB   r1, r1
  05: 44   ADD   r1, r0
```

The helper always emits four instructions for `r0` and six for `r1`–`r3`, so the cycle count stays predictable. It does clobber `N` and `Z`, because `SUB`, `ORI` and `SHL` all set flags, so set your registers up before a compare, never between the compare and the branch.

Cycle counts

The core is not pipelined and nothing is speculative, so a program's cycle count is the sum of the cycle counts of its instructions, and you can check it by hand.

instruction	cycles	
`ORI`	5	`CYCLE_1`…`CYCLE_5`, writes `r0` in `CYCLE_5`
`ADD SUB NAND SHL SHR LOAD`	4	write-back in `CYCLE_4`
`STORE JMP BZ BNZ BPZ`	3	retires in `CYCLE_3`
`0xC 0xD 0xE`	4	and no `done` pulse; see below

A branch costs 3 cycles whether or not it is taken.

> **Unimplemented opcodes: four cycles**
>
> These opcodes match no clause in `CYCLE_3`, so the FSM takes its default exit to `IDLE`, spends a cycle there, and is sent back to `CYCLE_1` by `run`, which is still asserted. When we measured four filler bytes in a row, they took 16 clock cycles, produced 0 `done` pulses, and had `halt` high on 4 of the 16.
>
> That last figure matters. `0x0C` is what `sp.start()` fills unwritten RAM with, so a program that runs off its own end makes the `halt` pin blink. "`halt` must stay low while `run` is asserted" is an assertion Lab 05 asks you to write, so if it fires in that situation, your test is working correctly on a machine that is behaving as designed.
>
> `golden.py` said three cycles until somebody measured it, and The golden model tells that story.

Writing a program

There are three ways to write a program, shown here from least to most readable:

```
prog = [0x2F, 0x44, 0xD9]              # ... which of these is the loop?
```

```
from simproc_asm import ORI, ADD, STORE
prog = [ORI(5), ADD(1, 0), STORE(1, 2)]
```

```
from simproc_asm import assemble
prog = assemble("""
        ORI   5              # r0 = 5
        ADD   r1, r0         # r1 = 5
loop:   SUB   r1, r0         # r1 -= 5
        BNZ   loop           # ... until it hits zero
        STORE r1, r2
""")
```

The text front end works out the branch displacements for you. That is the only reason to use it, but it's a good one, because almost everybody measures a displacement from the wrong place the first time.

`listing(prog)` prints a dump with the address, hex byte and mnemonic of each instruction. If you put it in your failure messages, it will save you twenty minutes when a test fails:

```
  00: 06   SUB   r0, r0
  01: 0F   ORI   1
  02: A6   SUB   r2, r2
```

Traps

Each of these has cost somebody an evening.

1. `ORI` ORs its immediate into `r0` and never clears any bits. Do `SUB r0, r0` first, or you get the OR of everything `r0` has ever held.
2. `ORI` always writes `r0`. There is no register field to get wrong, and no way to make it write anything else.
3. Shift amounts are 0–3. `SHL(0, 4)` raises an error rather than quietly truncating the amount: `shift amount 4 does not fit in 2 bits. SimProc shifts by 0, 1, 2 or 3 — to shift by 4, use two instructions.`
4. The branch range is `-8` to `+7`, measured from the instruction after the branch.
5. Arithmetic wraps without any warning. `200 + 60` is `4`, and there is no carry flag to check.
6. Addresses above 63 don't exist. A read from one returns `X` and a write to one is dropped, and neither of them aliases into low memory.
7. `LOAD rA, rB` and `STORE rA, rB` take the data register first and the pointer second. The order is the same for both, and it is the opposite of what many readers expect.
8. `SUB` is your compare, and it destroys `rA`. If you need the value afterwards, copy it first, and remember that a copy costs two instructions and overwrites both flags.

Links

- SimProc in one page: the machine, the FSM, the pins, the protocol
- The SimProc harness: what the test harness gives you
- `labs/common/simproc_asm.py`: this page as executable code, plus the disassembler
- `labs/common/golden.py`: this page as a reference model