# Lecture 11
# How assembler works

Computing platforms

Novosibirsk State University
University of Hertfordshire

D. Irtegov, A.Shafarenko

2018

# CdM-8 Opcodes

| S0 =0 | Fun 2-op | Rs | Rd |
|---|---|---|---|

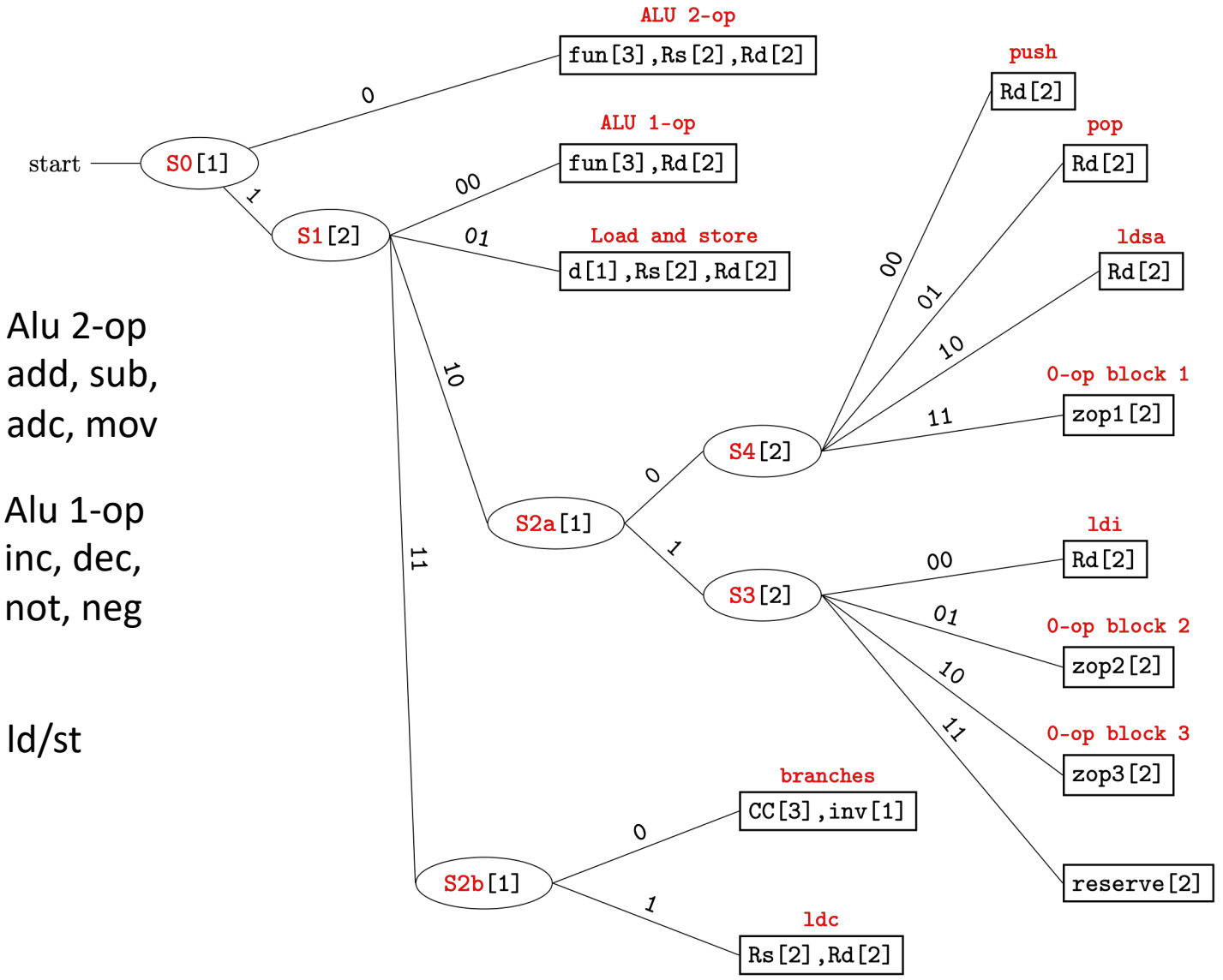
Alu 2-op
add, sub,
adc, mov

| S0 =1 | S1=00 | Fun 1-op | Rd |
|---|---|---|---|

Alu 1-op
inc, dec,
not, neg

| S0 =1 | S1=01 | d | Rs | Rd |
|---|---|---|---|---|

ld/st



# Full list of CdM-8 instructions

| bit-string (dec equiv) | fun (ALU 2-op) | fun (ALU 1-op) | d | zop1 | zop2 | zop3 | CC(inv=0) branches | CC(inv=1) |
|---|---|---|---|---|---|---|---|---|
| 0 | move | not | st | addsp | halt | ioi | beq | bne |
| 1 | add | neg | ld | setsp | wait | rti | bhs/bcs | blo/bcc |
| 2 | addc | dec | | pushall | jsr | crc | bmi | bpl |
| 3 | sub | inc | | popall | rts | osix | bvs | bvc |
| 4 | and | shr | | | | | bhi | bls |
| 5 | or | shla | | | | | bge | blt |
| 6 | xor | shra | | | | | bgt | ble |
| 7 | cmp | rol | | | | | br | nop |

Immediate operands for: `ldi`, `ldsa`, `jsr`, `branches`, `addsp`, `setsp`

# How CdM-8 assembler work

- Two passes
  1. Allocation pass
     - For each line of code, determine
     - Is this line labelled? (yes -> rememver label)
     - Is this line an instruction or dc/ds directive? (yes -> advance * by size of bit-string)
  - No actual code is generated on this pass,
    only lengths of bit-strings are calculated
  1. Generation pass
     - For each line, substitute values for labels and calculate expressions
     - Generate bit-strings for instructions and instruction operands
     - Generate bit-strings (values) for dc directives
     - Generate zero-flled bit-string for ds directve

# Why two passes?

- Because labels can be referenced before they are defined

```
        dec r0
        jle done
                # loop body
done:
        # continue after loop body
```

-

# Single-pass assemblers

- For every line of code
  - Line has a label? (yes -> remember it in symbol table)
  - Label mentioned in cross-reference table?
    (yes -> scan all references and substitute a value)
  - Line references a label? (yes -> remember it in cross-reference table)
    - Is referenced label already defined?
      (yes -> substitute value of the label)
      (no -> allocate a placeholder)
  - Generate code or data, probably using placeholders
- Single-pass assemblers are faster, but more complex
- And they consume more memory
- they need to store code with placeholders

# Linkers

- Conceptually, assembler+linker are similar to two-phase single-pass assembler
- Assembler compiling a code with external (unresolved) references must emit some code
- But it cannot emit finished code.
- It must use placeholders for references to external and relocatable labels
- And it must build a cross-reference table for every external label
- And it must build a cross-reference (relocation) table for every relocatable label

# CdM-8 object file (listing and file itself)

```
                    1             asect   0xe0
e0: 03              2   my>       dc 3
                    3   q>
e1: d2 e1           4             ldi r2,q
                    5             rsect   foo
00: 10              6   bar>      add r0,r0
01: d4              7             halt
                    8             rsect main
00: 71              9   main>     cmp r0,r1
01: e8 04          10             bhi z3
03: d5             11             wait
04: d4             12   z3:       halt
                   13             end
```



```
ABS  e0: 03 d2 e1
NTRY q e1
NTRY my e0
NAME main
DATA 71 e8 04 d5 d4
REL  02
NTRY main 00
NAME foo
DATA 10 d4
REL
NTRY bar 00
```

# In CdM-8, object files contain no tables

- Just lists of symbols and references
- And hexadecimal representation of code, data and placeholders
- So they are easy to read and easy to parse by Python
- “Real” computing platforms use binary object files
- Symbol and cross-reference tables are actual tables with headers, binary values and offsets

# Multi-pass assemblers

- In some Platform 2 (ISA), instructions can have variable length
- For example, branch instruction can have several forms:
  - With byte offset for address (can branch +127 bytes forward or 128 back)
  - With 16-bit offset
  - With 32-bit offset
  - With 64-bit offset
- x86/x64 ISA is example of such Platform 2
- When assembler compiles such instruction, It cannot know which form to use, so it must allocate longest possible placeholder
- But when it finds a label definition, it can select a shorter form
- But then all labels defined after this instruction - … - !!!

# Why multi-pass?

- Because, after you select shorter form for one branch instruction, you might find that you can select a shorter form for some other branches
- So, you must reassemble the program until no shorter form for every label-referencing instruction can be found
- Usually, two or three passes are sufficient, but for big program you might need more passes
- For external labels, assembler must use longest possible form in any case.