Lecture 10
Macros

Computing platforms

Novosibirsk State University
University of Hertfordshire

D. Irtegov, A.Shafarenko

2018

Repeating command sequences

```
ldi r0, label
ld r0, r0
```

- This sequence is repeated in many programs over and over
- In some ISA (CISC processors), there is a single command for this,
- But CISC hardware is much more complex
- Can we make a single [pseudo]instruction in software?

Yes we can

```
macro ldv/2
        ldi $1, $2
        ld $1, $1
mend
```

- ldv is macro (pseudo instruction) name
- /2 is number of parameters
- $1 and $2 are parameters

```
        ldv r0, label
```

Macro (macro definition)

- Macroprocessing is essentially a text substitution

  ldv r0, label

- $1=r0, $2=label

```
macro ldv/2
        ldi $1, $2
        ld $1, $1
mend
```

```
# macro ldv/2
        ldi r0, label
        ld r0, r0
# mend
```

Macroprocessing

- Substituting macros happens before or during compilation
  - Unlike subroutine call which happens at runtime
- Logically, it is always happens before compilaton,
- because result of macro substitution is compiled like normal code
- But the result of substitution can contain other macros…
- Many macro languages do not support recursive macros
- And practically none allow forward macro reference
- I.e. you always must define a macro before it can be used (unlike label or subroutine)

Why use macros?

- To make your program shorter (== easier to read)
- To make names for common idioms (instruction sequences)
- To avoid copy-paste code reuse
- Why copy-paste code reuse is bad?
  - If you find error in copypasted code,
  - you must find all copies
  - and fix them separately

Names for common idioms

```
macro clr/1. # clear a register
        xor $1,  $1
mend
macro test/1. # set Z and N flags according to register values
        mov $1, $1
mend
macro bnle/1
        bgt $1
mend
```

How not to use macros

```
macro incmem/1
        ldi r0, $1
        ld r0, r1
        inc r1
        st r0, r1
mend
```

- Looks good, but uses two registers
- And we cannot avoid this

How not to use macros (continue)

```
macro incmem/1 # “safe” version
        push r0
        push r1
        ldi r0, $1
        ld r0, r1
        inc r1
        st r0, r1
        pop r1
        pop r0
mend
```

Why it is a bad idea?

```
incmem a
incmem b
```

- Result of this macro substitution would have
  - two extra push and
  - two extra pop
- No good for a machine with 256 bytes of memory!
- Compilers often have so called peephole optimization
  - finding redundant commands in compiled code and removing them
- But assemblers usually literally assemble anything you wrote

So, macro is not universal tool

- It has strong limitations
- Sometimes, people are saying “compilers are just advanced macroprocessors”, but this is not correct
- Compilers (Level 4 platforms) are much more complex entities than macroprocessors
- But macros are simple and powerful (if used with judgement)
- There are pretty complex languages implemented as macros (LaTeX for example)

Nonce (apostrophe after a label name)

```
macro strcpy/2
            push r2
            push $1
            push $2
loop':      ld $1,r2
            inc $1
            st $2,r2
            inc $2
            move r2,r2
            bne loop'
            pop $1
            pop $2
            pop r2
mend
```

What if one of parameters will be r2?

```
unique $1,$2,temp
push ?temp
```

- Unique directive selects a register which is different from $1 and $2
- temp will be local symbol inside of the macro definition
- ? designates reference to such a symbol
- If we use ?temp instead of r2 in strcpy macro,
  it won't conflict with parameters
- Unique is not the only way to generate such symbols

Wait, there is more!

- Actually, CdM-8 “3 1/2” constructs, like if and while, are macros

```
if
   cmp r0,r1
is le
   move r0,r2
else
  move r1,r2
fi
```

```
      cmp r0,r1
      bgt else
      move r0,r2
      br done
else:
      move r1,r2
done:
```

But how???

- It is easy to prove that you cannot implement if-is-fi
  by simple text substitution (even with help of nonce and unique)
- To implement if-is-fi macros, you need to
  - Invent an unique name for a label
  - Remember it (somehow transfer it between is and fi macro definitions)
  - For if-is-else-fi, you need two unique labels
  - And how do you do nested if?

Nonce and unique are not enough!

```
macro if/0
        mpush _’
mend
macro is/1
        mpop id
        mpush alt?id
        bn$1  alt?id
mend
macro else/0
        mpop where
        mpush new?where
        br new?where
?where:
mend
macro fi/0
        mpop term
?term:
mend
```

How it all works?

- mpush and mpop are operations on *macro stack*
- *Macro stack* is a LIFO memory existing at compilation time
- mpush and mpop are directives, not instruction mnemonics
- mpush _’ engages a nonce and pushes it to stack

Let’s see this on the example

```
macro if/0                      # MS is short for [top of] Macro Stack
        mpush _’                # MS=_1
mend
macro is/1
        mpop id                 # id=_1
        mpush alt?id            # MS=alt_1
        bn$1  alt?id
mend
macro else/0
        mpop where              # where=alt_1
        mpush new?where         # MS=newalt_1
        br new?where
?where:                         # label will be alt_1
mend
macro fi/0
        mpop term               # term=newalt_1 or alt_1 (without else)
?term:                          # label will be newalt_1 or alt_1
mend
```

But what about loops?

- Loops have break/continue which can be used inside of the if blocks
- Break and continue cannot just pop the label from top of the stack
- To deal with this, loops use second macro stack,
  referenced by 1mpush and 1mpop directives
- Actually, there are three macro stacks in CdM-8 assembler

Another pair of useful macros

- Save and restore

```
save r1
  save r2
    save r3
        do something with r1,r2,r3
    restore
  restore
restore
```

- Much harder to restore registers in wrong order!