НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ ТА СПЕЦІАЛІЗОВАНИХ КОМП’ЮТЕРНИХ СИСТЕМ

**Лабораторна робота №4  
з дисципліни «Комп’ютерна схемотехніка»**

**Варіант 11**

Виконав  
студент 3-го курсу  
групи КВ-41  
Курач Віктор

Київ – 2017

**Постановка задачі**

Реалізувати методом емуляції системи команд мікрооперацію:

, – INT, – REAL

Кожній команді Асемблера має відповідати мікропрограма в ПМК. Перед виконанням операції операнд, який подано в форматі INT, необхідно перетворити в формат REAL. Кінцевий результат операції має бути подано в форматі REAL.

Операнди і слід розмістити в регістрах віртуального процесора.

**Розміщення команд в оперативній пам’яті**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Адреса ОП | КОП | REGx | REGy | Позначення операції |
| 100h | C1 | 1 | - | CONV REGa |
| 101h | A1 | 2 | 2 | MULF REGb, REGb |
| 102h | A1 | 2 | 1 | MULF REGa, REGb |
| 103h | FF | - | - |  |

**Структура мікропрограми**

ROZM \ розміщення в ОП команди Асемблера, операндів, CT, PSW

S1 SUBROUT 1 \ читання PC з ОП, команди за адресою з PC,  
\ розпаковування команди, читання операндів

{ jmap; or nil, r2, z; oey; } \ перехід на мікропрограму за кодом операції

S2 SUBROUT2 \ читання PSW з ОП, модифікація та запис нового PSW в  
\ ОП, запис результату в ОП, модифікація PC

org 0C10h

MCCONV \ мікропрограма перетворення з INT в REAL

org 0A10h

MCMULF \ мікропрограма множення REAL на REAL

org 0FF0h

MCFINISH \ мікропрограма закінчення роботи

END {}

**Лістинг програми**

macro mov reg1, reg2: { or reg1, z, reg2; }

\ Puprosements of BC1 registers:

\ R0 - PSW's address in RAM

\ R1 - PC's address in RAM

\ R2 - operation code

\ R3 - not used

\ R4 - number of REGy register

\ R5 - number of REGx register

\ R6 - PC's value

\ R7, R8 - buffers in subprograms

\ R9 - counter or buffer in subprograms

\ R10 - REGy's value

\ R11 - REGx's value

\ R12 - PSW's value

\ R13 - first argument for subprogram

\ R14 - second argument for subprogram

\ R15 - subprogram's result

\ RQ - not used

ROZM {}

accept r0: 0bh \ PSW's address in RAM

accept rdm\_delay: 2 \ RAM's speed

link l1: ct

link l2: rdm \ notRDM = 0 -> RAM is ready for next loop

link ewh: 16 \ dividing RgA by two parts

link m: 7, 6, 5, 4, 3, 2, 1, 0, z, z, z, z

dw 0h: 100h \ PC := 100h

dw 1h: 0fffch \ first operand's value

dw 2h: 03a0h \ second operand's value

dw 0bh: 0105h \ PSW

dw 100h: 0c111h \ C1 = operation code, 1 = REGx, 1 = REGy

dw 101h: 0a122h \ A1 = operation code, 2 = REGx, 2 = REGy

dw 102h: 0a112h \ A1 = operation code, 1 = REGx, 2 = REGy

dw 103h: 0ff22h \ FF = operation code, 2 = REGx, 2 = REGy

SUBROUT1 {}

\ 1) reading PC from RAM

S1 {}

{ xor r1, r1; oey; ewh; } \ forming higher levels of RAM addr reg

{ or nil, r1, z; oey; ewl; } \ lower levels

{ xor r6, r6; }

P1 { cjp rdm, P1; r; or r6, bus\_d, z; } \ R6 := PC's value

\ 2) reading instruction from RAM

{ or nil, r6, z; oey; ewl; } \ writing PC to RgA

{ xor r4, r4; }

P2 { cjp rdm, P2; r; or r4, bus\_d, z; } \ R4 := Assembler instruction

\ 3) instruction unpacking

{ xor r2, r2; }

{ or r2, r4, z; }

{ push nz, 7; }

{ rfct; or sr.0, r2, r2, z; } \ R2 := operation code

{ xor r5, r5; }

{ or r5, r4; }

{ and r4, 0fh; } \ R4 := 000 REGy

{ and r5, 0f0h; } \ R5 := 00 REGx 0

{ push nz, 3; }

{ rfct; or sr.0, r5, z; } \ R5 := 000 REGx

\ 4) reading operands from RAM

{ or nil, r4, z; oey; ewl; } \ RgA := 0000 REGy

{ xor r10, r10; }

P3 { cjp rdm, P3; r; or r10, bus\_d, z; } \ R10 := REGy's value

{ or nil, r5, z; oey; ewl; } \ RgA := 0000 REGx

{ xor r11, r11; }

P4 { cjp rdm, P4; r; or r11, bus\_d, z; } \ R11 := REGx's value

\ SUBROUT1 ends

\ 5) go to subprogram

\ operation code = subprogram's address in memory of microinstructions

\ (stored on local bus at this moment)

{ jmap; or nil, r2, z; oey; }

SUBROUT2 {}

\ 6) reading PSW from RAM; PSW's address stored in R0

S2 {}

{ or nil, r0, z; oey; ewl; } \ RgA := R0 value (PSW's address)

{ xor r12, r12; }

P5 { cjp rdm, P5; r; or r12, bus\_d, z; } \ R12 := PSW's value

\ 7) PSW's modification

{ cjp rn\_v, D1; }

{ and r12, 0fbffh; } \ overflow flag := 0

{ cjp nz, J1; }

D1 {}

{ or r12, 0400h; } \ overflow flag := 1

J1 {}

{ cjp rn\_c, D2; }

{ and r12, 0fffeh; } \ carry flag := 0

{ cjp nz, J2; }

D2 {}

{ or r12, 0001h; } \ carry flag := 1

J2 {}

{ cjp rn\_n, D3; }

{ and r12, 0feffh; } \ sign flag := 0

{ cjp nz, J3; }

D3 {}

{ or r12, 0100h; } \ sign flag := 1

J3 {}

{ load rm, flags; and nil, r15, 00ffh; } \ get result's mantissa

{ cjp rm\_z, D4; }

{ and r12, 0ffbfh; } \ zero flag := 0 (result != 0)

{ cjp nz, J4; }

D4 {}

{ or r12, 0040h; } \ zero flag := 1 (result = 0)

J4 {}

\ write PSW to RAM

{ or nil, r0, z; oey; ewl; } \ forming lower levels of RgA

RR1 { cjp rdm, RR1; w; or nil, r12, z; oey; }

\ write result to RAM; result is written to REGx which address is in R5

{ or nil, r5, z; oey; ewl; } \ forming lower levels of RgA

RR2 { cjp rdm, RR2; w; or nil, r15, z; oey; }

\ 8) PC modification (stored in R6; PC's address stored in R1)

{ add r6, 1; } \ PC := PC + 1

{ or nil, r1, z; oey; ewl; } \ RgA := 00000

RR3 { cjp rdm, RR3; w; or nil, r6, z; oey; } \ writing PC to RAM

{ cjp nz, S1; }

\ SUBROUT2 ends

org 0c10h

MCCONV {}

\ initial installations

{ mov r13, r11; } \ r13 := argument

{ xor r14, r14; } \ for result in direct code

{ xor r15, r15; } \ for result in REAL format

{ xor r7, r7; } \ for result's order

{ xor r8, r8; } \ for buffer

{ xor r9, r9; } \ for counter

{ or nil, r13, 0; load rn, flags; }

{ cjp rn\_z, C5; } \ if argument = 0, go to end

\ 1) transforming argument to direct code

{ or nil, r13, 0; load rn, flags; }

{ cjp not rn\_n, C1; }

{ mov r14, 8000h; } \ r14 := 1000 0000 0000 0000

{ mov r8, r13; } \ r8 := argument

{ xor r8, 0ffffh; } \ r8 := not r8

{ add r8, 1; } \ r8 := r8 + 1

{ or r14, r8; } \ r14 := - r8

{ cjp nz, C2; }

C1 {}

{ mov r14, r13; }

C2 {}

\ 2) writing down result's sign

{ mov r15, r14; }

{ and r15, 8000h; } \ r15[15] := result's sign

\ 3) calculating result's order

{ mov r8, r14; }

{ and r8, 7fffh; } \ r8 := (r14 < 0) ? -r14 : r14

{ mov r9, 10h; }

{ push; } \ sla buffer, until 1 in senior level detected

{ sub r9, 0; }

{ loop no; or sla, r8, 0; }

{ add r7, r9, nz; } \ r7 := result's order

{ mov r8, r7; }

{ push nz, 7; }

{ or sla, r8, 0; } \ buffer := sla order (8 times)

{ rfct; }

{ or r15, r8; } \ r15[13-8] := buffer

\ 4) calculating result's mantissa

{ mov r8, r14; }

{ and r8, 7fffh; } \ r8 := (r14 < 0) ? -r14 : r14

{ mov r9, r7; }

{ and nil, r9, 0fff8h; load rn, flags; }

{ cjp rn\_z, C4; }

{ sub r9, 8, nz; load rn, flags; }

{ cjp rn\_z, C3; }

{ push; }

{ or srl, r8, 0; } \ if order > 8, srl buffer (order-8) times

{ loop nz; sub r9, 0; }

{ cjp nz, C5; }

C3 {}

{ and r8, 0ffh; } \ if order = 8, r8 := 0000 0000 [mantissa]

{ cjp nz, C5; }

C4 {}

{ sub r9, 8, r9, nz; }

{ push; }

{ or sll, r8, 0; } \ if order < 8, sll buffer (8-order) times

{ loop zo; sub r9, 0; }

C5 {}

{ or r15, r8; load rn, flags;}

\ back to main program

{ cjp nz, S2; }

\ MCCONV ends

org 0a10h

MCMULF {}

\ initial installations

{ mov r13, r11; } \ r13 := first operand

{ mov r14, r10; } \ r14 := second operand

{ xor r15, r15; } \ for result

{ xor r7, r7; } \ for buffer

{ xor r8, r8; } \ for buffer

{ xor r9, r9; } \ for buffer

\ 1) forming result's sign

{ and r7, r13, 8000h; } \ r7[15] := 1st argument's sign

{ and r8, r14, 8000h; } \ r8[15] := 2nd argument's sign

{ xor r7, r8; }

{ mov r15, r7; } \ r15[15] := result's sign

\ 2) orders' addition

{ and r7, r13, 7f00h; } \ r7[14-8] := 1st argument's order

{ and r8, r14, 7f00h; } \ r8[14-8] := 2nd argument's order

{ add r7, r8; }

{ or r15, r7; } \ r15[14-8] := result's order

\ 3) mantissas' multiplication

{ and r7, r13, 0ffh; } \ r7[7-0] := 1st argument's mantissa

{ and r8, r14, 0ffh; } \ r8[7-0] := 2nd argument's mantissa

{ push nz, 7; }

{ or sla, r8, 0; }

{ rfct; } \ r8[15-8] := 2nd argument's mantissa

{ xor r9, r9; }

{ push nz, 6; }

{ or nil, r8, 7fffh; load rn, flags; }

{ cjp not rn\_n, M1; }

{ add r9, r7; } \ if Y[15] = 1, Z := Z+X

M1 {}

{ or sll, r9, 0; } \ Z := 2Z

{ or sll, r8, 0; } \ Y := 2Y

{ rfct; }

{ push nz, 7; }

{ or srl, r9, 0; }

{ rfct; } \ r9[7-0] := result's mantissa

\ 4) mantissa's normalization

{ xor r7, r7; }

{ xor r8, r8; }

M2 {}

{ and nil, r9, 80h; load rn, flags; }

{ cjp not rn\_z, M3; }

{ add r8, 1; } \ +1 order's error, if r9[7] = 0

{ or sll, r9, 0; } \ r9 := r9 \* 2

{ cjp nz, M2; }

M3 {}

{ push nz, 7; }

{ or sll, r8, 0; }

{ rfct; } \ r8[15-8] := mantissa's error

{ sub r15, r8, nz; } \ r15[14-8] := corrected result's order

{ or r15, r9; load rn, flags; }

\ back to main program

{ cjp nz, S2; }

\ MCMULF ends

org 0ff0h

FINISH { cjp nz, END; }

END {}

**Приклади роботи**



