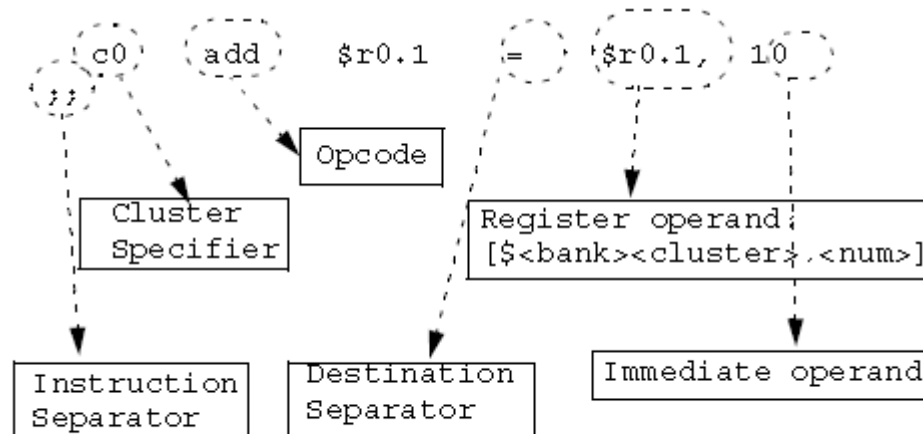


Assembly code generation for VEX

VEX is a clustered architecture. Multiple clusters can execute multiple statements in a single clock cycle. Since the ACSE generates the IR with the assumption that it is sequential, we did not attempt to use multiple clusters neither did we try to run more than one instruction in a single clock cycle.

Here is a sample instruction of the VEX architecture



Anatomy of the assembler notation for a VEX instruction

All the instructions generated in the IR had to be changed into the format as shown in the figure above. All instructions were mapped to cluster c0 and every instructions is preceded by a double semicolon (;;) .

Mapping of Instructions

Segments (DS, TEXT)

The VEX architecture supports 3 segments: DS, BSS and TEXT segments. The DS of the MACE is mapped to the data segment of the VEX. Appropriate changes have been made to accommodate the `.space` and the `.word` directives to map to `.comm` and `data4` directives in VEX. The text segment in VEX has the dummy main function and all the other instructions generated by ACSE. The BSS is used for uninitialized data, but since this is not present in MACE it is not generated for VEX.

Main Function (DUMMY)

ACSE 1.0 has an intermediate representation of code which does not include support for functions. The VEX assembly is built from C files and it is must to include the code in a main file. To accommodate this, changes were made in the function `translateCodeSegment()` . A dummy main function is added in the assembly file which acts as an encapsulation for the rest of the code. A stack pointer is initialized for the dummy main and used for `printf` and `scanf` function calls. The control also returns from the main function when the code in the IR reaches a `HALT`.

Register Mapping:

There are 56 registers available in VEX: R0.0 – R0.56. R0.0 has been hardwired to 0 as in the case of MACE. R0.1 is used for storing the stack pointer. After the IR is produced every instruction from R1 to R31 in MACE has been mapped to registers in Vex with an offset. R1-R32 are R0.4-R0.35 in Vex. R0.3 has been used for passing a parameter in the function printf and scanf.

Arithmetic operations:

Most of the arithmetic operations could be mapped with the MACE architecture. There were some that could not be matched. It would have taken a considerable effort to accommodate the same with existing instructions available in VEX. We have stressed more on making the basic code running on a VEX platform.

Branch Operations:

The branch operation in VEX requires the use of special branch registers as the branching is not done based on some flags like the CC or the Z flag. Every SET operation has been changed to update a branch register so that the branches can be taken on BEQ and BNE instructions. Since branch registers are necessary for branching, branching on a condition without using a SET operation is not possible. At this state most of the code generated in ACSE uses a conditional branch with a SET operation before it. As BT is an unconditional jump operation it has been mapped to goto statement in VEX.

Load Store Operations:

Load and store operations are very similar to that of MACE. Store operations in Vex require the destination operand to be first. This is opposite to that generated in ACSE. Hence Reg-1 is printed second and the label for store operation is printed first.

Labels:

There is only a minor difference in the way labels are used in MACE and in Vex. Necessary changes have been made to make it work.

I/O Operations

READ instruction has been mapped to scanf function for Vex. To map the scanf function a separate string is added to the data segment area of the code after the end of main. This string is taken as the parameter for the scanf function. The register that is passed for scanning does not have the value of the scan on return. The value of the register gets corrupted, but the scan value is stored in the memory that is represented by the register. Since all calculations in ACSE are done on the same variable, a dummy label is created in the data segment, just to hold the value of the scan. After the scan function is over this value is copied back into the register.

WRITE instruction has been mapped to printf function in Vex. The process for successfully executing the printf is very similar to that of scanf.

An example of code generation: for MACE and VEX

```
int a,b;  
a = 0;  
  
read(b);  
  
for(a = 0; b>10; a=a+1) {  
    b = b - 1;  
}  
  
write(a);
```

Fig 1: A simple code in ACSE

```
.data  
L0: .WORD 0  
L1: .SPACE 12  
.text  
MOVA R1 L1  
ADDI R2 R0 #15  
ADD (R1) R0 R2  
MOVA R2 L1  
ADDI R2 R2 #1  
ADDI R1 R0 #10  
ADD (R2) R0 R1  
MOVA R1 L1  
ADDI R1 R1 #2  
ADDI R2 R0 #20  
ADD (R1) R0 R2  
ADDI R2 R0 #0  
ADDI R2 R0 #0  
STORE R2 L0  
L2: MOVA R1 L1  
ADDI R1 R1 #2  
ADD R3 R0 (R1)  
SUBI R10 R3 #10  
SGT R10 0  
BNE L4  
BEQ L3  
L5: LOAD R2 L0  
ADDI R4 R2 #1  
ADDI R2 R4 #0  
STORE R2 L0  
BT L2  
L4: MOVA R5 L1  
ADD R6 R0 (R5)  
SUBI R7 R6 #1  
MOVA R8 L1  
ADD (R8) R0 R7  
MOVA R9 L1  
ADDI R9 R9 #1  
ADD R10 R0 (R9)  
SUBI R11 R10 #1  
MOVA R12 L1  
ADDI R12 R12 #1  
ADD (R12) R0 R11  
MOVA R13 L1  
ADDI R13 R13 #2  
ADD R14 R0 (R13)  
SUBI R15 R14 #1  
MOVA R16 L1  
ADDI R16 R16 #2  
ADD (R16) R0 R15  
BT L5  
L3: LOAD R2 L0  
WRITE R2 0  
STORE R2 L0  
HALT
```

Fig 2: Code Generated in MACE with the IR of ACSE (Fig 1)

```

.comment " Created at ALaRI "
.section .bss
.align 32
.section .data
.align 32
L0:: .data4 0
L1:: .data4 0
LX:: .data4 0
.section .text
.proc
.entry caller, sp=$r0.1, rl=$l0.0, asize=-32, arg()
main::
.trace 1
c0 add $r0.1 = $r0.1, (-0x20)
c0 add $r0.12 = $r0.0,0
;;
c0 mov $r0.2 = (_?1SP.1 + 0)
;;
c0 stw 0x10[$r0.1] = $l0.0
;;
c0 mov $r0.11 = (LX + 0)
;;
c0 stw 0x10[$r0.1] = $l0.0
;;
.call scanf, caller, arg($r0.2:u32,$r0.11:u32), ret($r0.2:s32)
c0 call $l0.0 = scanf
;;
c0 mov $r0.2 = $r0.0
;;
c0 ldw $l0.0 = 0x10[$r0.1]
xnop 3
;;
c0 ldw $r0.11 = ((LX+0)+0)[$r0.0]
;;
c0 stw L1[$r0.0] = $r0.11
;;
c0 add $r0.12 = $r0.0,0
;;
c0 stw L0[$r0.0] = $r0.12
;;
L2:
c0 ldw $r0.11 = L1[$r0.0]
;;
c0 sub $r0.13 = $r0.11,10
;;
c0 stw L1[$r0.0] = $r0.11
;;
c0 cmpgt $b0.1 = $r0.13, 0
;;
c0 br $b0.1, L4
;;
c0 brf $b0.1, L3
;;
L5:

// Continued in block 2

```

```

L5:
c0 ldw $r0.12 = L0[$r0.0]
;;
c0 add $r0.13 = $r0.12,1
;;
c0 add $r0.12 = $r0.13,0
;;
c0 stw L0[$r0.0] = $r0.12
;;
c0 goto L2
;;
L4:
c0 ldw $r0.11 = L1[$r0.0]
;;
c0 sub $r0.14 = $r0.11,1
;;
c0 add $r0.11 = $r0.14,0
;;
c0 stw L1[$r0.0] = $r0.11
;;
c0 goto L5
;;
L3:
c0 ldw $r0.12 = L0[$r0.0]
;;
c0 mov $r0.2 = (_?1SP.1 + 0)
;;
c0 stw 0x10[$r0.1] = $l0.0
;;
c0 stw (LX+0)[$r0.0] = $r0.12
;;
.call printf, caller, arg($r0.2:u32,$r0.12:s32), ret($r0.2:s32)
c0 call $l0.0 = printf
;;
c0 mov $r0.2 = $r0.0
;;
c0 ldw $l0.0 = 0x10[$r0.1]
xnop 3
;;
c0 ldw $r0.12 = ((LX+0)+0)[$r0.0]
;;
c0 stw L0[$r0.0] = $r0.12
;;
.return ret($r0.2:s32)
c0 return $r0.1 = $r0.1, (0x20), $l0.0
;;
.endp
.section .data
_?1SP.1:
.data1 37
.data1 100
.data1 0
.section .text
.import printf
.type printf,@function
.import scanf
.type scanf,@function

```

Fig 3: Code Generated for Vex with the IR of ACSE(Fig 1)