NTU Compiler Technology Project 4

tags: `NTU_CT` `NTU` `Compiler Techonology of Programming Language` `Code Generation`

How to run?

First, put your testing file and `main.S` to the folder same as `parser` file

```
$ make
$ ./parser [testing file].c
$ aarch64-linux-gnu-gcc -static -O0 main.S
$ qemu-aarch64-static ./a.out
```

Or, you can put the `run.sh` and `main.S` in the same directory and run

```
$ ./run.sh [your parser] [the c file]
$ qemu-aarch64-static ./a.out
```

Note that you must change your test file's `main()` to `MAIN()`

Implementation

Just show the mainly implementation concept

Assignment statements

```
void genAssignStmt(AST_NODE *assignNode)
{
    AST_NODE *it = assignNode;
    unpack(it, id, relop_expr);
    REG reg = genExprRelated(relop_expr);
    genVariableAssign(id, reg);
    freeReg(reg);
}
...
void genVariableAssign(AST_NODE *idNode, REG val)
```

```
{
    assert ( idNode->nodeType == IDENTIFIER_NODE );
    assert ( getIDEntry(idNode) != NULL );
    assert ( getIDAttr(idNode)->attributeKind == VARIABLE_ATTRIBUTE );
    TypeDescriptor* typeDescriptor = getIDTypeDescriptor(idNode);

    if(getIDKind(idNode) == ARRAY_ID){
        genArrayAssign(idNode, val);
        return;
    }else{
        REG addr;
        if(getIDGlobal(idNode)){
            addr = getReg();
            fprintf(output, "ldr x%d, =_g_%s\n", addr, getIDName(idNode));
        }else{
            int offset = getIDOffset(idNode);
            addr = genIntLiteral(offset);
            fprintf(output, "sub x%d, x29, x%d\n", addr, addr);
            fprintf(stderr, "Var assign name: %s, offset: %d\n",
getIDName(idNode), offset);
        }

        if(idNode->dataType == INT_TYPE){
            fprintf(output, "str w%d, [x%d, #0]\n", val, addr);
        }else{
            fprintf(output, "str s%d, [x%d, #0]\n", val, addr);
        }

        freeReg(addr);
    }
    return;
}
```

Arithmetic expressions

```
REG genRelopExpr(AST_NODE *exprNode)
{
    AST_NODE* it = exprNode->child;

    if (isConstExpr(exprNode))
        return genConstValue(exprNode);

    if (getExprKind(exprNode) == BINARY_OPERATION) {
        unpack(it, lvalue, rvalue);

        REG LReg = genExprRelated(lvalue);
        REG RReg = genExprRelated(rvalue);

        if(lvalue->dataType == INT_TYPE && rvalue->dataType == INT_TYPE){
            switch(getExprOp(exprNode)){
                case BINARY_OP_ADD:
                    fprintf(output, "add w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_SUB:
                    fprintf(output, "sub w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_MUL:
                    fprintf(output, "mul w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_DIV:
                    fprintf(output, "sdiv w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;

                case BINARY_OP_EQ:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, eq\n", LReg);
                    break;
                case BINARY_OP_GE:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, ge\n", LReg);
                    break;
                case BINARY_OP_LE:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, le\n", LReg);
                    break;
                case BINARY_OP_NE:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    break;
                case BINARY_OP_GT:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, gt\n", LReg);
                    break;
                case BINARY_OP_LT:
                    fprintf(output, "cmp w%d, w%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, lt\n", LReg);
                    break;

                case BINARY_OP_AND:
                    fprintf(output, "cmp w%d, #0\n", LReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    fprintf(output, "cmp w%d, #0\n", RReg);
                    fprintf(output, "cset w%d, ne\n", RReg);
                    fprintf(output, "and w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_OR:
                    fprintf(output, "cmp w%d, #0\n", LReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    fprintf(output, "cmp w%d, #0\n", RReg);
                    fprintf(output, "cset w%d, ne\n", RReg);
                    fprintf(output, "orr w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
            }
        }else{
            // Float expr
            if(lvalue->dataType == INT_TYPE)
                fprintf(output, "scvtf s%d, w%d\n", LReg, LReg);

            if(rvalue->dataType == INT_TYPE)
                fprintf(output, "scvtf s%d, w%d\n", RReg, RReg);
            switch(getExprOp(exprNode)){
                case BINARY_OP_ADD:
                    fprintf(output, "fadd s%d, s%d, s%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_SUB:
                    fprintf(output, "fsub s%d, s%d, s%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_MUL:
                    fprintf(output, "fmul s%d, s%d, s%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_DIV:
                    fprintf(output, "fdiv s%d, s%d, s%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_EQ:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, eq\n", LReg);
                    break;
                case BINARY_OP_GE:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, ge\n", LReg);
                    break;
                case BINARY_OP_LE:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, le\n", LReg);
                    break;
                case BINARY_OP_NE:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    break;
                case BINARY_OP_GT:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, gt\n", LReg);
                    break;
                case BINARY_OP_LT:
                    fprintf(output, "fcmp s%d, s%d\n", LReg, RReg);
                    fprintf(output, "cset w%d, lt\n", LReg);
                    break;
                case BINARY_OP_AND:
                    fprintf(output, "fcmp s%d, #0\n", LReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    fprintf(output, "fcmp s%d, #0\n", RReg);
                    fprintf(output, "cset w%d, ne\n", RReg);
                    fprintf(output, "orr w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
                case BINARY_OP_OR:
                    fprintf(output, "fcmp s%d, #0\n", LReg);
                    fprintf(output, "cset w%d, ne\n", LReg);
                    fprintf(output, "fcmp s%d, #0\n", RReg);
                    fprintf(output, "cset w%d, ne\n", RReg);
                    fprintf(output, "orr w%d, w%d, w%d\n", LReg, LReg, RReg);
                    break;
            }
        }
        freeReg(RReg);
```

```
            return LReg;
        }else{
            // Unary operation

            unpack(it, value);
            REG reg = genExprRelated(value);

            if(value->dataType == INT_TYPE){
                switch(getExprOp(exprNode)){
                    case UNARY_OP_POSITIVE:
                        break;
                    case UNARY_OP_NEGATIVE:
                        fprintf(output, "neg w%d, w%d", reg, reg);
                        break;
                    case UNARY_OP_LOGICAL_NEGATION:
                        fprintf(output, "cmp w%d, #0\n", reg);
                        fprintf(output, "cset w%d, eq\n", reg);
                        break;
                }
            }else{
                switch(getExprOp(exprNode)){
                    case UNARY_OP_POSITIVE:
                        break;
                    case UNARY_OP_NEGATIVE:
                        fprintf(output, "fneg s%d, s%d", reg, reg);
                        break;
                    case UNARY_OP_LOGICAL_NEGATION:
                        fprintf(output, "fcmp s%d, #0\n", reg);
                        fprintf(output, "cset w%d, eq\n", reg);
                        break;
                }
            }
            return reg;
        }
    }
```

Control statements: while, if-then-else

```
void genWhile(AST_NODE *whileNode)
{
    AST_NODE *it = whileNode;
    unpack(it, test, stmt);
    int while_n = const_n++;

    fprintf(output, "_WHILE_%d:\n", while_n);
    if (test->nodeType == STMT_NODE && getStmtKind(test) == ASSIGN_STMT){
        genAssignStmt(test);
        test = test->child;
    }
    REG reg = genExprRelated(test);
    if (test->dataType == FLOAT_TYPE)
        fprintf(output, "fcvtzs w%d, s%d\n", reg, reg);

    fprintf(output, "cmp w%d, #0\n", reg);
    freeReg(reg);
    fprintf(output, "beq _WHILE_END_%d\n", while_n);
    genStmt(stmt);
    fprintf(output, "b _WHILE_%d\n", while_n);
    fprintf(output, "_WHILE_END_%d:\n", while_n);
}
...
void genIf(AST_NODE *ifNode)
{
    AST_NODE *it = ifNode;
    unpack(it, test, stmt, elseStmt);
    int if_n = const_n++;

    fprintf(output, "_IF_%d:\n", if_n);
    if (test->nodeType == STMT_NODE && getStmtKind(test) == ASSIGN_STMT){
        genAssignStmt(test);
        test = test->child;
    }
    REG reg = genExprRelated(test);
    if (test->dataType == FLOAT_TYPE)
        fprintf(output, "fcvtzs w%d, s%d\n", reg, reg);

    fprintf(output, "cmp w%d, #0\n", reg);
    freeReg(reg);
    fprintf(output, "beq _ELSE_%d\n", if_n);
    genStmt(stmt);
    fprintf(output, "b _END_IF_%d\n", if_n);


    fprintf(output, "_ELSE_%d:\n", if_n);
    //if (elseStmt->nodeType != NUL_NODE)
    genStmt(elseStmt);
    fprintf(output, "_END_IF_%d:\n", if_n);
}
```

Parameterless procedure calls

```
void genFunctionCall(AST_NODE *functionCallNode)
{
    AST_NODE *it = functionCallNode;
    unpack(it, id, param);

    char *name = getIDName(id);
    if (!strcmp(name, "write")){
        genWrite(functionCallNode);
    } else if (!strcmp(name, "read")){
        fprintf(output, "bl _read_int\n");
    } else if (!strcmp(name, "fread")){
        fprintf(output, "bl _read_float\n");
    } else {
        fprintf(output, "bl _start_%s\n", name);
    }
}
```

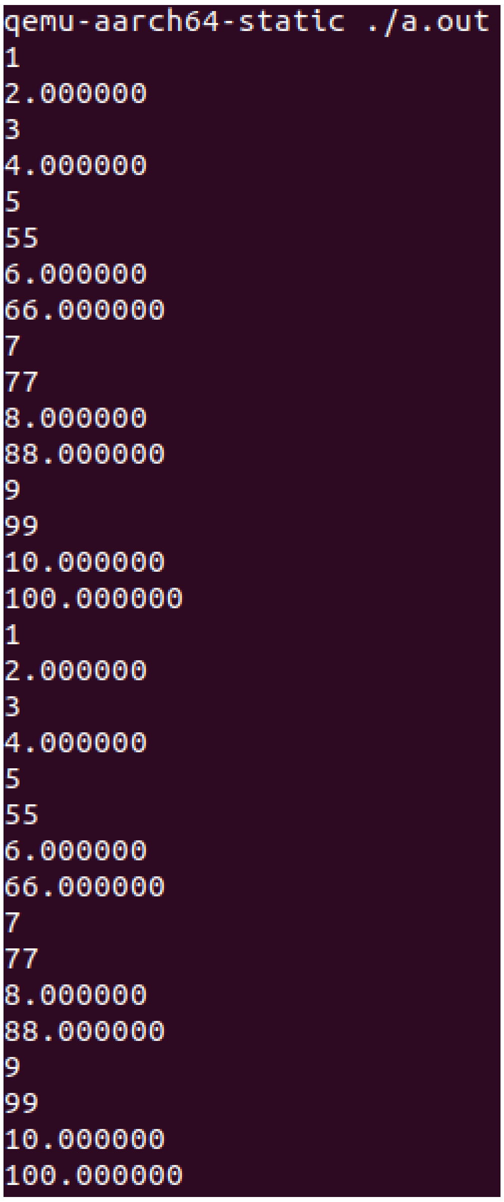
Read and Write I/O calls

```
void genWrite(AST_NODE *functionCallNode){
    AST_NODE *it = functionCallNode;
    unpack(it, id, paramList);
    AST_NODE *param = paramList->child;

    REG reg = genExprRelated(param);
    switch(param->dataType){
        case INT_TYPE:
            fprintf(output, "mov w0, w%d\n", reg);
            fprintf(output, "bl _write_int\n");
            break;
        case FLOAT_TYPE:
            fprintf(output, "fmov s0, s%d\n", reg);
            fprintf(output, "bl _write_float\n");
            break;
        case CONST_STRING_TYPE:
            fprintf(output, "mov x0, x%d\n", reg);
            fprintf(output, "bl _write_str\n");
            break;
    }
    freeReg(reg);
}
```

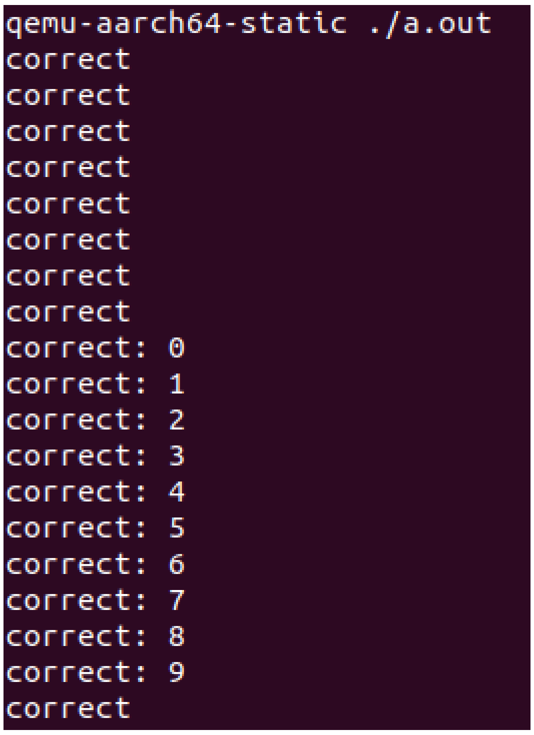
Experience result

- **assign.c**



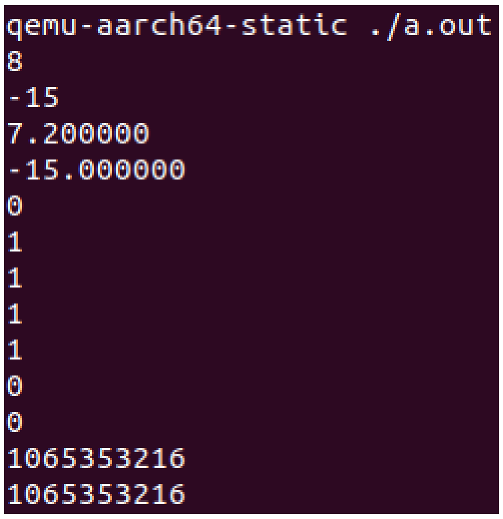
```
qemu-aarch64-static ./a.out
1
2.000000
3
4.000000
5
55
6.000000
66.000000
7
77
8.000000
88.000000
9
99
10.000000
100.000000
1
2.000000
3
4.000000
5
55
6.000000
66.000000
7
77
8.000000
88.000000
9
99
10.000000
100.000000
```

- **control.c**



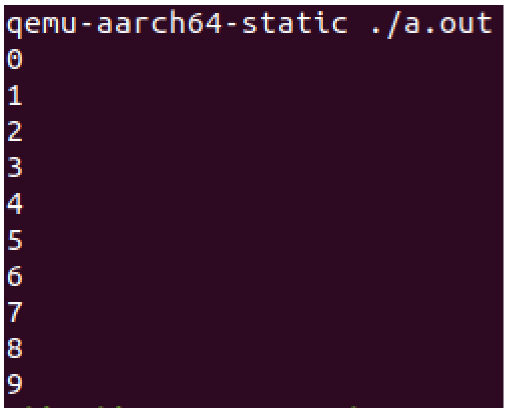
```
qemu-aarch64-static ./a.out
correct
correct
correct
correct
correct
correct
correct
correct
correct: 0
correct: 1
correct: 2
correct: 3
correct: 4
correct: 5
correct: 6
correct: 7
correct: 8
correct: 9
correct
```

- **expr.c**



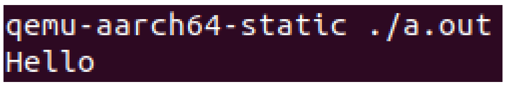
```
qemu-aarch64-static ./a.out
8
-15
7.200000
-15.000000
0
1
1
1
1
0
0
1065353216
1065353216
```

- **func.c**



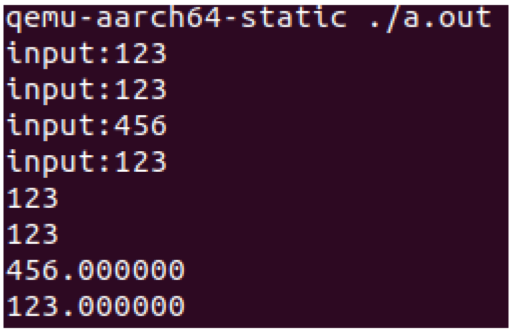
```
qemu-aarch64-static ./a.out
0
1
2
3
4
5
6
7
8
9
```

- **hello.c**



```
qemu-aarch64-static ./a.out
Hello
```

- **io.c**



```
qemu-aarch64-static ./a.out
input:123
input:123
input:456
input:123
123
123
456.000000
123.000000
```