CMSC 430 Code Generation Notes
M. Zelkowitz
4/21/99

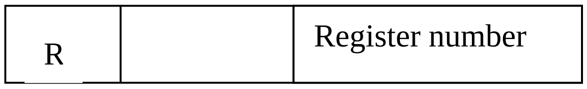
This is the basic design of an interpretive code generator. The basic ideas are:

- to create a code pattern for each quad operator.
- Translate high level programming language operators such as *, +, iftest, … into low level machine language operators such as loadregister, addregister, …

Address words

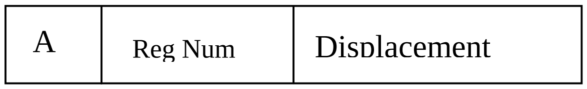
This code generator is based upon an address word as the basic data structure. Address words come in 2 flavors: register address words (RAW) and address words (AW).

RAW: This signifies that the data resides in a register. The address word format is:

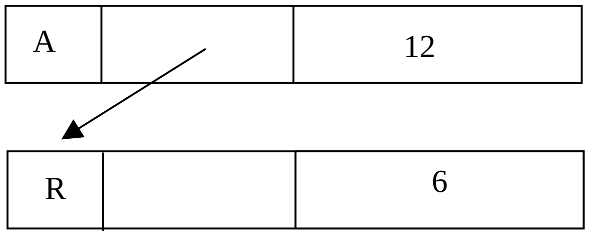


Thus we have register words for register 0, 1, 2, 3, …8, 9.

AW: This signifies that the data is in memory a certain displacement after the contents of a register. The format is:

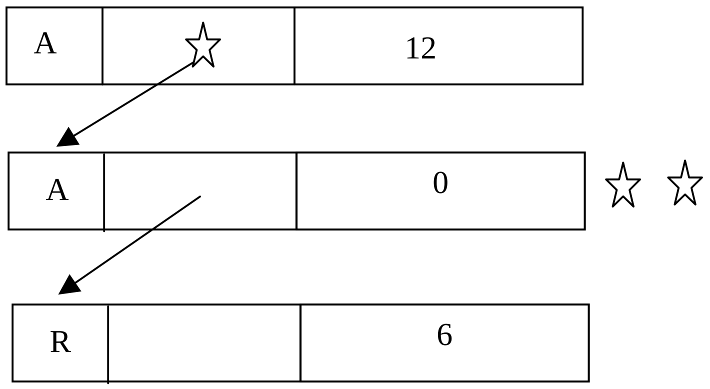


This is usually used to represent information in an activation record. The register part points to the register containing the activation record pointer, and the displacement part points to the offset past that activation record. Thus if register 6 points to the local activation record, and variable X is at word 12 in that activation record, the data structure that represents that address will be:



When code is generated, if X is to be loaded in a register, then the AW for X is passed to the code generator. First the code generator makes sure that the RegNum field of A is in a register. In this case it is, so the effective address of index register 6 with displacement of 12 is formed, or instruction part 60012 is returned.

Assume a variable is one level back in declaration. Then the register pointer for X will point to the static link pointer in the activation record. The data structure for this will then be:



In order for the pointer to the activation record to be in a register (the * in the above figure) the ** must be converted from an A to a R. But this is given as 0 words past the register address word for register 6 (The static link in the current activation record).

So in order to access X in this case, the code generator must first (assuming register 5 is always used for temporary index register calculations) generate:

```
            Load 5,6,0000   (01560000)
```

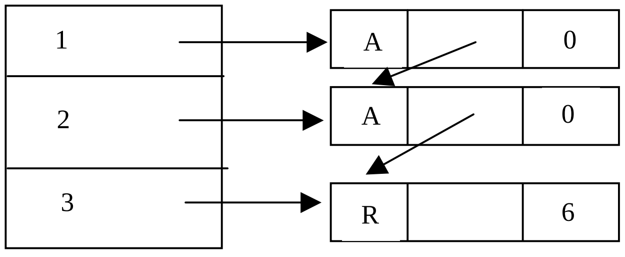
Then the effective address is now 12 words past register 5 or 50012.

Code generator design

On entry to each procedure (the <procstart> quad), the code generator must set up a vector of Aws to represent accessing each activation record. This if we have:

```
Procedure P
          Procedure Q
                    Procedure R
```

Then the code generator must set up upon entry to R the following ARLINK array. ARLINK[1] corresponds to variables in P, ARLINK[2] tp variables in Q, and ARLINK[3] to variables in R:



ARLINK

This says that the act rec pointer for R (local proc) is at arlink[3], which means register 6. The act rec pointer for Q, is at ARLINK[2], which means follow static link at 0 past register 6. The act rec pointer for P is at RLINK[1] which means follow 0 past register 6 and then 0 past that pointer to get to activation record start for P.

Low level code generation

The code generator will have structure:

```
Procedure codegen(quadtype, arg1, arg2, result)
        Do case(quadtype)

                Quadop1: …

                Quadop2: …

                Quadop3: …
                ,,,
        end
```

For each quadtype, produce a pattern of code.

Step 1: arg1, arg2 and result are symbol table addresses. Convert each to a address word:

```
Procedure codegen(quadtype, arg1, arg2, result)
        CreateAW(arg1,Awarg1);
        CreateAW(arg2,Awarg2);
        CreateAW(result,Awresult);
        Do case(quadtype)
                 Quadop1: …
                Quadop2: …
                Quadop3: …
                ,,,
        end
```

CreateAW creates data structure:

A	ARLINK [nesting level]	Act Rec offset from variable's symbol table entry

At this point we have address words for each variable.

Genword(8digits) – generates 8digit instruction at location LC and LC=LC+1. For example, if LC=0123, then genword(01560112) will output:

```
        0123 08 01560112
```

and set LC=0124. (See HAC430 loader format if confused at this point.)

Genopcode(op,reg,ind,flag,disp) will output the instruction:

```
        Op_reg_ind_flag_disp
```

For example, following the genword(01560012) example above, if the next function is genopcode(01,0,5,false,0014), genopcode will create the 8digit object 01050014 and do a genword(01050014) which will cause:

```
        0124 08 01050014
```

to be output.

Mid-Level operations

Geninst(inst,raw,aw) – generates inst register,data
It uses AW and RAW to generate intermediate instructions

Thus for: quadtype of plus, the code sequence could be:

```
        Geninst(load, R0, Awarg1);
        Geninst(add,R0,Awarg2);
        Geninst(store,R0,Awresult);
```

If Awarg1 is a local variable at act rec location 12 and if Awarg2 is a variable one level back at act rec location 14 and Awresult is a local variable at act rec location 16, then the code generated is:

```
        01060012  -- load arg1 into R0
        01560000  -- Follow static link to get arg2
        03050014  -- add arg2 to R0
        02060016  -- store sum in result
```

Problem with this code generator is that it does redundant loads and stores. See code for (A+B+C) and for (A+B)+(C+D).

We can make code generator "smarter" by adding the following additional functions:

Allocreg(RAW X) -- Find a register that is free and set its RAW in X. Initially R5, R6, and R7 have special uses. All the other 7 can be so allocated.

Freereg(RAW X) – Set register X as now free.

Setreg(AW X, RAW Y) – Set argument X to currently be in register Y so the next time it X is needed, register Y can be used instead.

Ifreg(AW X, RAW Y) – If X is now in a register, return that register name as Y. Otherwise, allocate a register, load X into that register, and return its name as Y.

With these 4 new instructions, we can now generate good code:

```
        Plus: ifreg(Awarg1,AW1)
              Geninst(Add,AW1,Awarg2)
              Setreg(Aw1,Awresult)

        Assign: ifreg(Awarg1,AW1)
              Geninst(store,AW1,Awresult)
              Freereg(AW1)
```

Show that for D:= (A+B+C) (all local variables) we generate:

```
        Load R0,A
        Add R0,B
        Add R0,C
        Store R0,D
```

And for E:=(A+B)+(C+D) we now get:

```
        Load R0,A
        Add R0,B
        Load R1,C
        Add R1,D
        AI R0,(R1) –How to add one register  to another
        Store R0,E
```

These code sequences are optimal without additional code improvement strategies.

Questions:
1. What if allocreg has no registers to allocate?
2. Do you understand add contents of regA to regB?
3. How to handle function parameters?
4. How to handle labels and forward and backward jumps?