

The Black Art of Code Generation

aka
Some Assembly Required



by Walter Bright
walterbright@x.com
Dconf Sep 2024

Textbooks On

- Lexing 1,000
- Parsing 100
- Semantics 10
- Optimizing 1
- Code Generation 0

A black rectangular sign with rounded corners and a white border is mounted on two weathered wooden posts. The sign features the words "code" and "generator" in a white, lowercase, sans-serif font, stacked vertically. The sign is positioned on a rocky, gravelly slope with patches of dry, orange-brown grass. In the background, a vast mountain valley unfolds under a heavy, grey, overcast sky. The valley floor is filled with a dense forest of evergreen trees. In the distance, several mountain peaks are visible, some with patches of snow or light-colored rock. The overall scene is a rugged, high-altitude landscape.

Gotta Rely On

- Articles
- Papers
- CPU Specification Documentation
- Dogged Persistence
- Godbolt.org
- Walter Bright

This Will Be About DMD's Code Generator

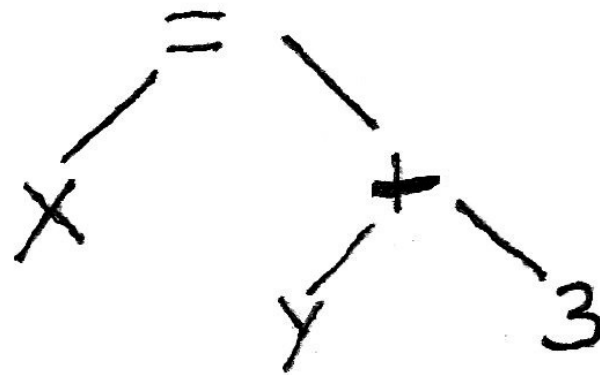
- I know next to nothing about the Gnu or LLVM code generators
- It is not derived from anybody else's code generator
- It started as an 8088 code generator, and survived upgrades to 32 bits, 64 bits, and SIMD
- AFAIK it is a unique design
- BOOST licensed, so anyone can use it for any purpose

Data Structures

Understanding the code generator's data structures is key

Expressions

$$x = y + 3$$



Expression Tree Structure

```
struct elem {  
  
    ubyte Eoper;  
    tym_t Ety;  
    eflags_t Eflags;  
    ubyte Ecount;  
    ubyte Ecomsub;  
  
    union {  
        constant  
        variable  
        E1,E2  
    }  
}
```



```
int horse(int x, int y) {  
    return x = y + 3;  
}
```

Compile with `--b` switch:

```
x(1) = (y(0) + 3L );  
x(1);
```

Compile with `--b --f`:

```
el:0x1af1740 cnt=0 cs=0 , TYint 0x1af16d0 0x1af14a0  
el:0x1af16d0 cnt=0 cs=0 = TYint 0x1af14a0 0x1af1660  
el:0x1af14a0 cnt=0 cs=0 var TYint x  
el:0x1af1660 cnt=0 cs=0 + TYint 0x1af1580 0x1af15f0  
el:0x1af1580 cnt=0 cs=0 var TYint y  
el:0x1af15f0 cnt=0 cs=0 const TYint 3L  
el:0x1af14a0 cnt=0 cs=0 var TYint x
```

Block Structure

```
struct block {  
    ubyte BC;        // BCgoto/BCiftrue/BCret/BCretexp  
    elem* Belem;    // expression tree for this block  
    list_t Bsucc;   // next block(s) to execute  
    block* Bnext;  // next block in function  
    code* Bcode;   // generated code  
}
```

```
int popCount(uint x) {  
    int n = 0;  
    while (x) {  
        n += x & 1;  
        x >>= 1;  
    }  
    return n;  
}
```

1: BCgoto
 n = 0;
 Bsucc: B2

2: BCtrue
 x;
 Bsucc: B3 B5

3: BCgoto
 n += (x & 1);
 Bsucc: B4

4: BCgoto
 x >>>= 1;
 Bsucc: B2

5: BCretexp
 n;

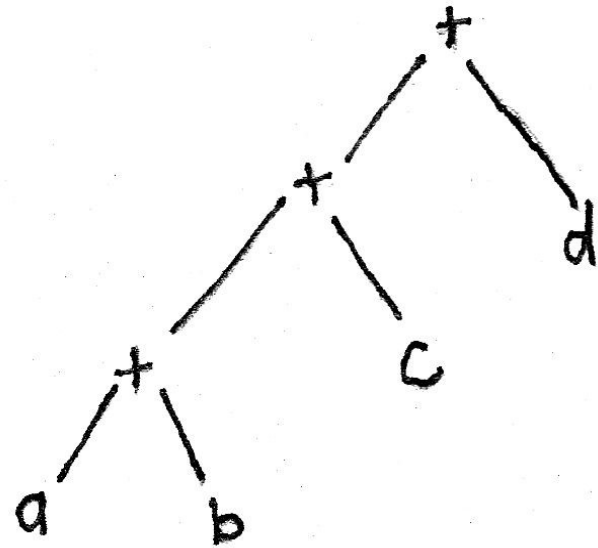
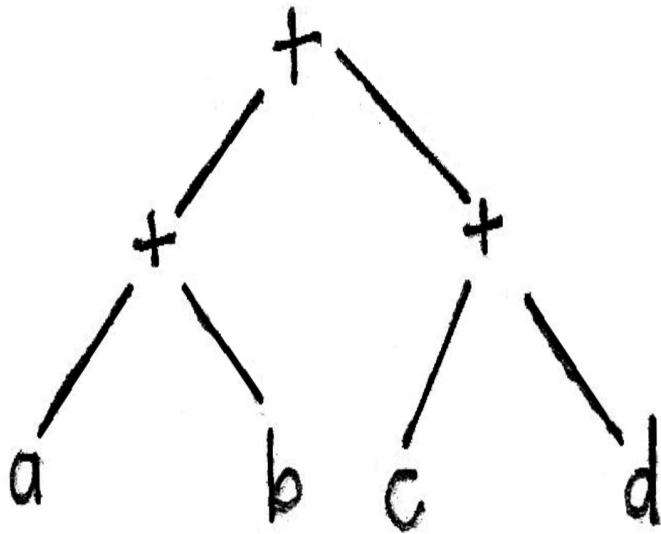
Code Structure

```
struct code {  
    code* next;  
    code_flags_t iflags;  
    uint lop;  
    uint lea;  
    FL IFL1;  
    evc IEV1;  
    FL IFL2;  
    evc IEV2;  
}
```

_D5test18popCountFkZi:

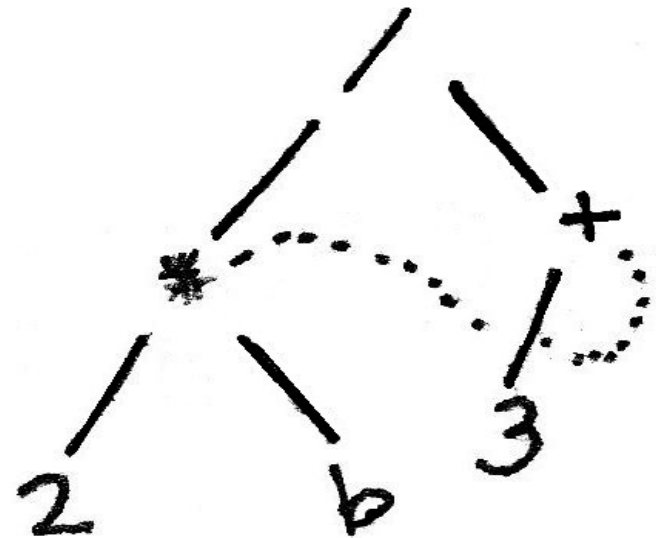
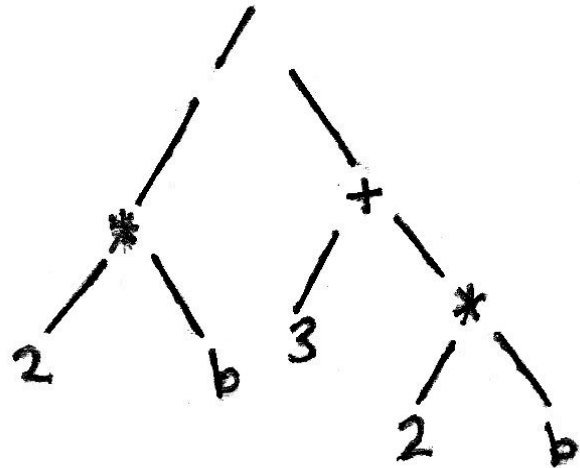
0000:	55							push	RBP		
0001:	48	8B	EC					mov	RBP,RSP		
0004:	48	83	EC	10				sub	RSP,010h		Prolog
0008:	89	7D	F8					mov	-8[RBP],EDI		
000b:	C7	45	F0	00	00	00	00	mov	dword ptr -010h[RBP],0		B1
0012:	83	7D	F8	00				cmp	dword ptr -8[RBP],0		B2
0016:	74	10						je	L28		
0018:	8B	45	F8					mov	EAX,-8[RBP]		
001b:	25	01	00	00	00			and	EAX,1		B3
0020:	01	45	F0					add	-010h[RBP],EAX		
0023:	D1	6D	F8					shr	dword ptr -8[RBP],1		B4
0026:	EB	EA						jmp	short L12		
0028:	8B	45	F0					mov	EAX,-010h[RBP]		B5
002b:	C9							leave			
002c:	C3							ret			Epilog

cgelem.d local optimizations



Needs one less register

cgcs.d Common Subexpressions



Register Masks

- `alias` `ulong regm_t`;
- `allregs` : integer registers
- `fpregs` : floating point registers
- `xmmregs` : XMM registers
- `mfuncreg` : registers preserved by function
- `msavereg` : registers we'd like to save

EA – Effective Address

- offset
- base register + offset
- base + index * scale + offset

Entry Point – `dout.writefunc()`

- Sets up symbol table (a simple array of symbols)
- Optionally runs the intermediate code optimizer
- Calls `codgen()` the code generator
- Shuts down data structures
- Emits debug info for function
- Saves result for possible inlining

Code Generator

- Sets the stage
- If optimized
 - Do the following in a loop
 - Generates code in depth-first order
 - Allocate unused registers to variables
 - Until no more registers can be allocated
- Else
 - Loops through block list generating code in that order

Code generator 2

- Compute locations of local variables
- Generate function prolog
- Generate function epilog
- Assign addresses in generated code
- Pinhole optimizations
- Optimize jumps
- Generate switch tables
- Emit generated code

One Function per Operator

```
// jump table
private immutable nothrow void function
  (ref CGstate, ref CodeBuilder, elem *,
   Ref regm_t)[OPMAX] cdxxx =
[
    OPunde:      &cderr,
    OPadd:       &cdorth,
    OPmul:       &cdmul,
    OPand:       &cdorth,
    OPmin:       &cdorth,
    OPnot:       &cdnot,
    OPcom:       &cdcom,
    OPcond:     &cdcond,
    ...
];
```

elem nodes are one of:

- Rvalue
 - Constant
 - Result of an elem node
 - Address of an lvalue
 - Common subexpression
 - Create it
 - Reuse it
- Lvalue
 - Variable
 - Indirection

```

void cdcom(ref CGstate cg, ref CodeBuilder cdb,
           elem *e, ref regm_t pretregs)
{
    if (pretregs == 0) {
        codelem(cgstate, cdb, e.E1, pretregs, false);
        return;
    }
    tym_t tym = tybasic(e.Ety);
    int sz = _tysize[tym];
    uint rex = (I64 && sz == 8) ? REX_W : 0;
    regm_t possregs = (sz == 1) ? BYTEREGS
                               : cgstate.allregs;
    regm_t retregs = pretregs & possregs;
    if (retregs == 0)
        retregs = possregs;
    codelem(cgstate, cdb, e.E1, retregs, false);
    getregs(cdb, retregs); // retregs are destroyed
}

```

```

const reg = (sz <= REGSIZE)
    ? findreg(retregs)
    : findregmsw(retregs);
const op = (sz == 1) ? 0xF6 : 0xF7;
genregs(cdb, op, 2, reg);          // NOT reg

code_orrex(cdb.last(), rex);
if (I64 && sz == 1 && reg >= 4)
    code_orrex(cdb.last(), REX);

if (sz == 2 * REGSIZE) {
    const reg2 = findreglsw(retregs);
    genregs(cdb, op, 2, reg2);    // NOT reg+1
}

fixresult(cdb, e, retregs, pretregs);
}

```


Register Allocator cgreg.d

- Optimizer computes “live range” of each variable, which is a bit vector of the basic blocks, with a bit set for each block a variable is live in
- Code generator sets a bit for each register used in each block
- A variable is mapped to a register if that register is not used in any of the blocks the variable is live in
- This is done in a loop until no more variables can be enregistered

Finishing Things up

- Write code bytes to buffer
- Write fixups
- Write object code to file
 - elfobj.d
 - machobj.d
 - mscoffobj.d
 - cgoobj.d (for Win32 OMF, obsolete)

What's a Fixup?

- Addresses of symbols cannot be resolved until the program is linked
- A fixup record that has the following:
 - Location of the fixup
 - Name of the symbol
 - Offset from that symbol
 - Relative or absolute
 - Tend to be confusing and very poorly documented

Inherent Limitations of This Code Generator

- Byte registers AH, BH, CH, DH, etc., are not independent registers
- Bit mask of 64 bits means no more registers can fit in them
 - (64 registers ought to be enough for anyone!)
- Cannot have loops inside of expression tree
 - Limits inlining possibilities
- x87 is a giant kludge

Why Is Code Gen So Complicated?

- 8 bit microprocessors had ~40 instructions
- AArch64 had over 2000 instructions each with variations
- Processor variations as CPUs progress
- Endless ways to put instructions together
- x87 has its own wacky way of doing things that doesn't fit



Loading a Const Into a Variable

- MOV EA, const
- MOV EA, register
 - Is the right value coincidentally already in a reg?
 - Is the constant a common subexpression in a reg?
 - Is this faster?
 - MOV register, const then MOV EA, register
- Multiple instructions needed?
- EA in a register?
- Affect on the flags?

A Code Generator is Never Finished

- Always new ideas on generating code appear
- CPUs constantly change
 - Optimal code sequences change
- Code generator author will never be unemployed!

But This Presentation is Finished!

