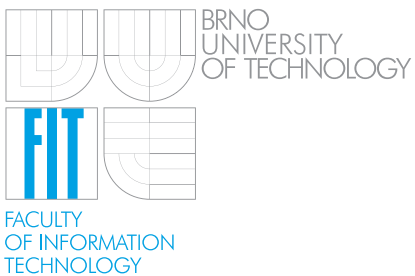
Code Generation: Intermediate Languages

Jakub Sochor Milan Pála

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 00 Brno, CZ
www.fit.vutbr.cz



VYPe 2012, Brno

Contents

Intermediate languages (IML)

Source code to intermediate language

- Rather then compile directly to binary
- Optimized for ideal computer – virtual machine
- Easy for testing
- Can be more optimized
- Several back ends (vs one parser)

Virtual machines

- Many registers
- Orthogonal instruction sets
- =>IML are larger, less efficient

Some forms of IML

Triples, 3-touples

- `(Operator source destination|target)`
- eg. `(ADD D0 D1)` == (d1 += d0)

Quadruples, Quads

- `(Operator destination|target source source)`
- (d = s1 + s2) == (+, d, s1, s2)
- (=, d, s, —)

Implicit assignment

Implicit assignment

- Instruction without explicit target (eg. LESS_THEN, ADD)

LESS_THEN

```
(LESS_THEN, a, b)
(GOTO, target, -)
```

ADD

- (+, B, C) == (A = B+C)

Triples vs Quads

Triples

- Close in structure to real assembly languages

Quadruples

- Compact: (+, d, s1, s2) vs (=, d, s1) (+, d, s2)
- Less position dependent

Reverse-Polish Notation (RPN, postfix)

Advantages

- No parentheses: (1+2) * (3+4) = 1 2 + 3 4 + *
- No temporary variables (only run-time stack)

RPN example

Example

stack	input
empty	1 2 + 3 4 + *
1	2 + 3 4 + *
1 2	+ 3 4 + *
3	3 4 + *
3 3	4 + *
3 3 4	+ *
3 7	*
21	empty

Contents

C-Code

- Intermediate language based on C macros
- No requirment for virtual machine
- Usage:

```
compiler file.code -o intermediate.c
gcc intermediate.c -o file
```

Infrastructure

- Data types:
  - 8-bit chars
  - 16-bit words
  - 32-bit long words
  - generic pointers
- 16 registers
- 2048-element stack (32b)
- frame pointer, stack pointer, instruction pointer

Infrastructure implementation

```
1  #define STACK_DEPTH 2048
2  typedef union reg {
3      char *pp;
4      lword l;
5      struct _words w;
6      struct _bytes b;
7  }
8
9  reg r0, r1, r2;
10
11 reg stack[STACK_DEPTH];
12
13 reg *__sp = &stack[STACK_DEPTH];
14 #define sp ((char *) __sp)
```

Addressing modes

- immediate – `28, 0xFF, 'c'`
- direct – `variable`
- effective-address – `&variable`
- indirect

```
1 W(p+offset) // word at offset from pointer p
2 W(r0.pp + r1.w.low)
3 r0.pp = WP(fp+8) // fetch pointer to word
4                  // at address fp+8
5 r1.w = W(r0.pp) // fetch the word
```

Flow control

```
1 if (r1.l == r2.l)
2 { code1 }
3 else
4 {  code2  }
```

```
1 EQ(r1.l, r2.l)
2    goto thenBranch;
3 elseBranch:
4    code2...
5    goto endif;
6 thenBranch:
7    code1...
8 endif:
```

Instructions implementation

```
1 #define EQ(a, b) if ((long)(a) == (long)(b))
```

```
1 #define push(val)   (--sp)->l = (lword)(val)
2 #define pop(type)  (type)((sp++)->l)
3
4 push( r0.w.low );
5 r1.w.low = pop(word)
```

Operators

C operators:	`= += -= *= /= %= ^= &= |=`
Arithmetic shifts:	`<<= >>=`
Two complement:	`=-`
One complement:	`=~`
Logic right shift:	`lrs(x,n)`

Subroutines

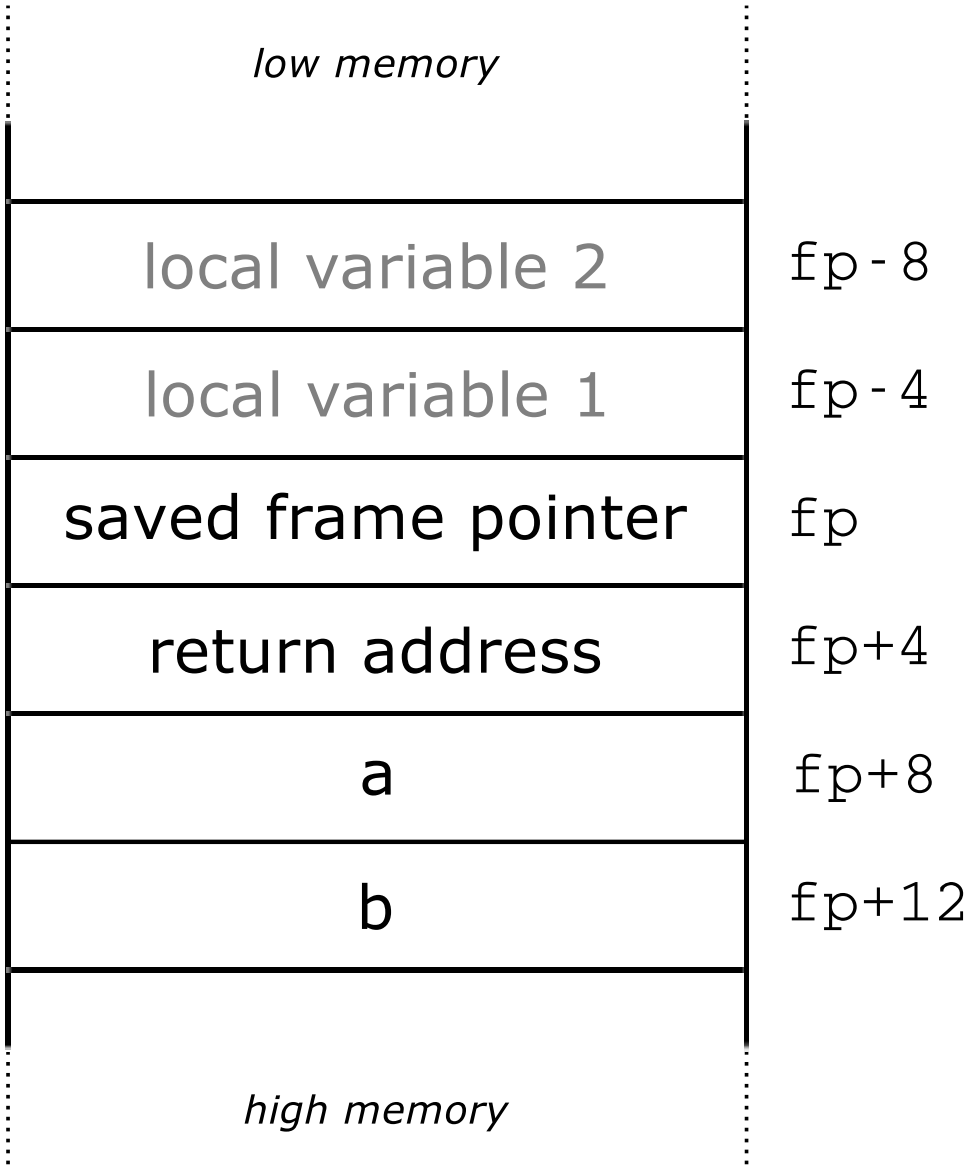
- Arguments pushed in reversed order
- Return address as part of the call
- Subroutine uses stack for local variables
- Return value in register `rF`
- Caller cleans stack after return from subroutine

Subroutine example

```
1 long sub(a,b) {
2     return a-b;
3 }
```

```
1 PROC(sub, public)
2     push(fp);
3     fp = sp;
4     rF.l = L(fp+8); // fetch a
5     rF.l -= L(fp+12); // decrement by b
6     sp = fp;
7     fp = pop(ptr);
8     ret(); // return value in rF
9 ENDP(sub)
```

Stack visualisation



Subroutine instructions implementation

```
1 #define PROC(name, cls)     cls name() {
2 #define ENDP(name)           ret(); }
3
4 #define call(name) (-- __sp)->pp = #name,\
5                    (*(void (*) ()) (name))   ()
6 #define ret()  __sp++; return
```

References

- HOLUB A.I.: *Compiler Design in C*. Prentice Hall, 1990, ISBN 0-13-155045-4.