Compiler Design Project Report

C to 8085 Assembly Code

February 21, 2018

Group Members

Saurabh Srivastava(201551032)
Sakshee Jain (201551074)
Sakshi Dubey (201551096)
Ayush Das (201551098)

Indian Institute of Information Technology

1 Introduction

1. The main focus of this project is to convert the given C language program to 8085 (assembly language) program.
2. The proposed model transforms :
   (a) arithmetic operations - addition, subtraction, multiplication and division.
   (b) unary operations
   (c) relational operations.
3. We have divided our project into two broad phase :
   (a) **First Phase :**
      i. Design grammar to parse the given C language program by taking only those operations which are supported in 8085 architecture.
      ii. Design and analyze 8085 architecture grammar.
      iii. Parsing a given small piece of C language program to check, whether the operations are supported in 8085 architecture or not.
   (a) **Second Phase :**
      i. Actual conversion to 8085 assembly program.

2 8085 Programming model

8085 processor has a set of seven 8-bit registers including the accumulator and six others, namely, B, C, D, E, H and L. Depending upon applications, the registers other than the accumulator can be used either as independent byte-registers or as 16-bit register pairs(e.g simultaneous combination of BC, DE, etc.).

3 Assumptions and Constraints

3.1 Constraints

1. Since 8085 model only accepts 8-bit data value in registers(that can only hold upto 8-bits) hence we have considered only 2 integer-based data types in C language, i.e int_8 and uint_8.
2. Currently we are ignoring numbers greater than 2^8.
3. The number of registers that we can use is fixed. They are 7 in total namely A,B,C,D,E,H and L including accumulator
4. 8085 microprocessor can handle 16 bits at the maximum. So,we can only use 2 registers at a time for any kind of operation.

3.2 Assumptions

We have taken a few assumptions in preparing the Grammar for our convenience. We have not considered the following types of statements :

1. Looping statements.
2. Procedures or Functions.
3. Modulus operation.

4 Grammar

1. <program> $\longrightarrow$ $< block >$
2. <block> $\longrightarrow$ $< declaration >< stmtlist >$
3. <type> $\longrightarrow$ $int8\_t \quad | \quad uint8\_t$
4. <declaration> $\longrightarrow$ $< type >< decl - init > \quad | \quad < type >< declaration - list >$
5. <declaration-list> $\longrightarrow$ $< decl - init >, < declaration - list > \quad | \quad < decl - init >$
6. <decl-init> $\longrightarrow$ $< id >; \quad | \quad < id >< equal - op >< assign >;$
7. <assign> $\longrightarrow$ $< simple - expr > | < variable >$
8. <stmt-list> $\longrightarrow$ $< stmt >< stmt - list > | \quad \varepsilon$
9. <stmt> $\longrightarrow$ $< stmt > | < comp - stmt > | < expr - stmt > | < break - stmt >$ | <exit-stmt> | <if-stmt>
10. <comp-stmt> $\longrightarrow$ $\{< block >\}$
11. <break-stmt> $\longrightarrow$ $break$
12. <expr-stmt> $\longrightarrow$ $< expr >; \quad | \quad ;$

13. <expr> $\longrightarrow$ $< id > + =< expr >$ | $< id > - =< expr >$ | $< id > * =< expr >$
| <id> /= <expr> | <id> ++ | <id> −− | $< simple - expr >$
| <id> = <simple-expr>

14. <simple-expr> $\longrightarrow$ $< simple - expr > or < and - expr > | < and - expr >$

15. <and-expr> $\longrightarrow$ $< and - expr > and < unary - rel - expr > | < unary - rel - expr >$

16. <unary-rel-expr> $\longrightarrow$ $< not >< unary - rel - expr > | < rel - expr >$

17. <rel-expr> $\longrightarrow$ $< add - expr >< rel - op >< add - expr > | < add - expr >$

18. <rel-op> $\longrightarrow$ <= | < | > | >= | == | !=

19. <add-expr> $\longrightarrow$ $< add - expr >< sum - op >< term > | < term >$

20. <sum-op> $\longrightarrow$ + | −

21. <term> $\longrightarrow$ $< term >< mul - op >< unary - expr >$ | $< unary - expr >$

22. <mul-op> $\longrightarrow$ $*$ | / | %

23. <unary-expr> $\longrightarrow$ $< unary - op > (< unary - expr >)$ | $factor$

24. <unary-op> $\longrightarrow$ −− | ++

25. <factor> $\longrightarrow$ $(< add - expr >)$ | $< variable >$

26. <if-stmt> $\longrightarrow$ $if(< condition >) < stmt >$
| if (< condition >)< stmt > else < stmt >

27. <variable> $\longrightarrow$ $< id > | < constants >$

28. <condition> $\longrightarrow$ $< condition >< rel - op >< factor > | < factor >$

29. <equal-op> $\longrightarrow$ =

30. <and> $\longrightarrow$ &&

31. <or> $\longrightarrow$ ||

32. <not>$\longrightarrow$!

5 Examples

1. int C = 0 ;

   $\langle type \rangle$ $\langle id >$ = $\langle id \rangle$;
   /* Load 0 to register using MVI */
2. int C = A + B ;
   $\langle type \rangle$ $\langle id >$ = $\langle sum\text{-}expr \rangle$
   /* compute ADDITION ADD A,B / * Load value of A to register using MOV
3. int C = A * 2 ;
   $\langle type \rangle$ $\langle id >$ = $\langle mul\text{-}expr \rangle$
4. int C = B - A ;
   compute Subtraction SUB A,B Load value of A to register using MOV
5. int C = 2 + 2 ;
   /* Load constant in registers A,B
   /* compute addition and store result i.e. ADD A,B
   /* Load value of A to register using MOV i.e MOV C A

6.
```
if(expression)
    {
        <stmt>
    }
```

   $\langle if\text{-}stmt \rangle \longrightarrow if\ (\langle\ condition\ \rangle)\langle\ stmt\ \rangle$

7.
```
if(expression)
{
   <stmt>
}
else
```

{
<stmt>
}

⟨*if-stmt*⟩ ⟶ *if* (⟨ *condition* ⟩)⟨ *stmt* ⟩ else ⟨ *stmt* ⟩

/* Compare using CMP two registers or register with accumulator
/* Jump to particular instruction line and start execution

6 SDT

1. **If B then S**

   — **B : id == id**

   B.false = gen Label()
   S.code = expr
   B.code = relop expr
   S.code =
   append(gen (B.code))
   append(JNE B.false)
   append(gen (S.code))
   append(B.false)

   — **B : id < id**

   B.false = gen Label()
   S.code =
   append(gen(B.code))
   append(JNC B.false)
   append(gen(S1.code))
   append(B.false :)

— **B : id $\geq$ id**

B.false = gen Label()
S.code =
append(gen(B.code))
append(JNC B.false1)
append(B.false1 : JNZ B.false)
append(gen(S1.code))
append(B.false :)

— **B : id $\leq$ id**

B.true = gen Label()
B.false = gen Label()
S.code =
append(gen(B.code))
append(JC B.true)
append(JNZ B.false)
append(B.true :)
append(gen(S1.code))
append(B.false :)

— **B : id $>$ id**

B.false = gen Label()
B.false2 = gen Label()
S.code =
append(gen(B.code))
append(JC B.false1)
append(B.false1 : JNZ B.false)
append(gen(S1.code))
append(B.false :)

2. **If (B) then S1 else S2**

— **B : id < id**

B.false = gen Label()
S.code =
append (gen(B.code))
append (JNC B.false)
append (gen(S1.code))
append (B.false :)
append (gen(S2.code))

— **B : id > id**

B.false = gen Label()
S.code =
append(gen(B.code))
append(JC B.false1)
append(B.false1 : JNZ B.false)
append(gen(S1.code))
append(B.false :)
append(gen(S2.code))

— **B : id == id**

B.false = gen Label()
S.code =
append(gen(B.code))
append(JNZ B.false)
append(gen(S1.code))
append(B.false :)
append(gen(S2.code))

— **B : id ≤ id**

B.true = gen Label()
B.false = gen Label()

S.code =
append(gen(B.code))
append(JC B.true)
append(JNZ B.false)
append(B.true)
append(gen(S1.code))
append(B.false :)
append(gen(S21.code))

— **B : id $\geq$ id**

B.false = gen Label()
S.code =
append(gen(B.code))
append(JNC B.false1)
append(B.false1 : JNZ B.false)
append(gen(S1.code))
append(B.false :)
append(gen(S2.code))

3. **Arithmetic Operations**

c = a + b ;
R1 = getReg(c)
R2 = getReg(a)
R3 = getReg(b)
gen("ADD R3 , R2 , R1")

4. **Conditions**

id relop id
R1 = getReg(id)
R2 = getReg(id)
B.code = gen(CMP R1 R2)

id relop id - where one id value is in accumulator
B.code = gen(CMP id)

7 Some Examples

1. **Example 1 :**

   int a=10 ;
   int b=3 ;
   int c=a+b ;

```
    LD #10
    ST $110

    LD #3
    ST $111

    MVI D 00
    MVI A 00
    MOV B $110
    MOV C $111
    LOOP:
    ADD B
    JNC NEXT
    INR D
    NEXT:
    DCR C
    JNZ LOOP
    LD $120
    ST $112

return  .end

```

2. **Example 2 :**

```
if(3>=5)
{
    int c=5;
}
else
{
    int d=4;
}
```

```
1       MOV A, #3
2       CMP A, #5
3       JNC bfalse1
4       LD #5
5       ST $110
6
7       JMP if1
8
9       bfalse1:
10      LD #4
11      ST $111
12
13  if1 NOP
14
15  return  .end
16
```

3. **Example 3 :**
   int a = 4 ;
   int b = 5 ;
   int c = a + b ;

```
    LD #4
    ST $110

    LD #5
    ST $111

    MOV A, $110
    MOV B, $111
    ADD A, B
    ST $120

    LD $120
    ST $112

return  .end

```

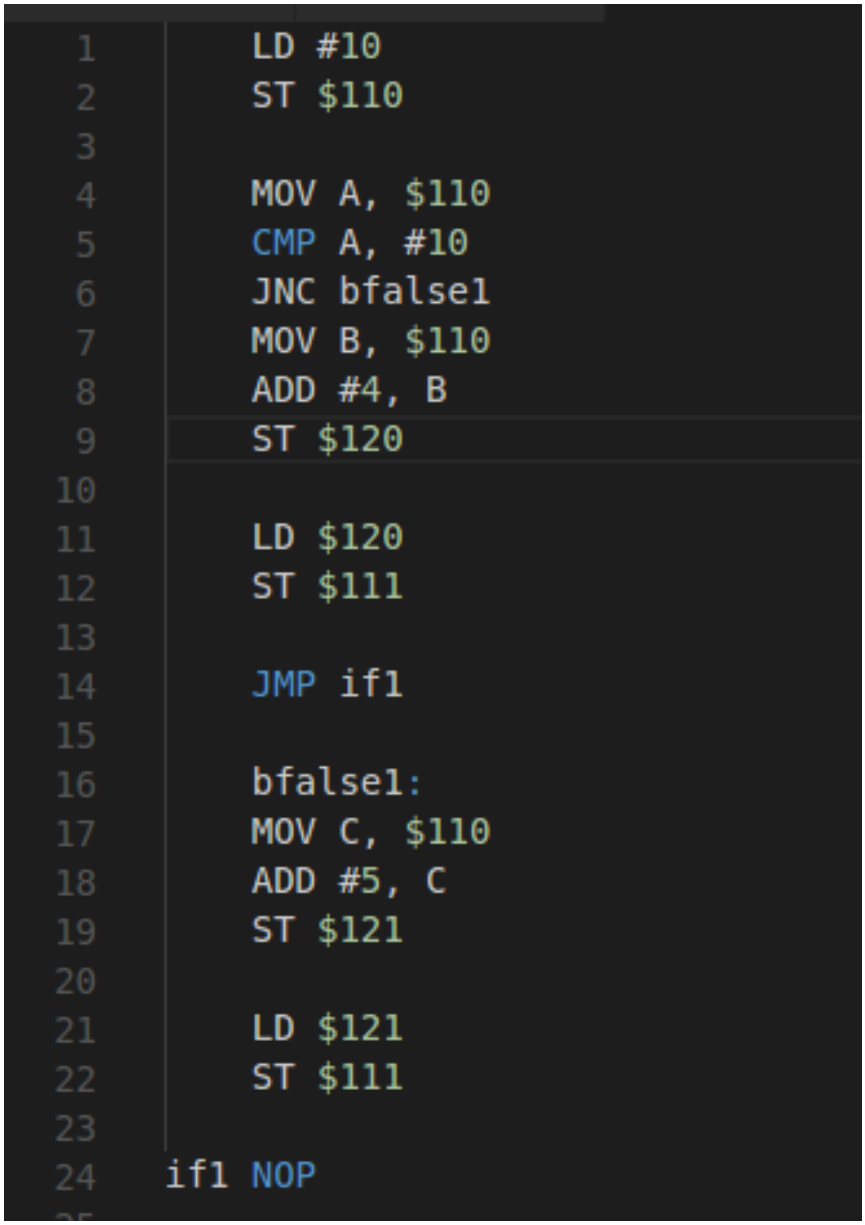
4. **Example 4 :**

```
if(3<4)
{
    int a=3;
}
else
{
    int d=5;
}
```

```
1       MOV A, #3
2       CMP A, #4
3       JNC if1
4
5       LD #3
6       ST $110
7
8       JMP if1
9
10      bfalse1:
11      LD #5
12      ST $111
13
14  if1 NOP
15
16  return  .end
17
```

5. **Example 5 :**

```
int a=10;
if(a>=10)
{
    int c=4+a;
}
else
{
    int c=5+a;
}
```



```
1       LD #10
2       ST $110
3
4       MOV A, $110
5       CMP A, #10
6       JNC bfalse1
7       MOV B, $110
8       ADD #4, B
9       ST $120
10
11      LD $120
12      ST $111
13
14      JMP if1
15
16      bfalse1:
17      MOV C, $110
18      ADD #5, C
19      ST $121
20
21      LD $121
22      ST $111
23
24    if1 NOP
```

6. **Example 6 :**

```
int a=10;
int c=a*2;
```

```
1       LD #10
2       ST $110
3
4       MVI D 00
5       MVI A 00
6       MOV B $110
7       MOV C #2
8       LOOP:
9       ADD B
10      JNC NEXT
11      INR D
12      NEXT:
13      DCR C
14      JNZ LOOP
15      LD $120
16      ST $111
17
18    return  .end
```

7. **Example 7 :**

```
int a=14;
int c=7;
int d=a*c;
```

```
1       LD #14
2       ST $110
3
4       LD #7
5       ST $111
6
7       MVI D 00
8       MVI A 00
9       MOV B $110
10      MOV C $111
11      LOOP:
12      ADD B
13      JNC NEXT
14      INR D
15      NEXT:
16      DCR C
17      JNZ LOOP
18      LD $120
19      ST $112
20
21  return  .end
```

8. **Example 8 :**

int a=10−4;

```
1       MOV A, #10
2       SUB A, #4
3       ST $120, A
4
5       LD $120
6       ST $110
7
8   return  .end
```

9. **Example 9 :**

```
int a=5;
if(a!=0)
{
    a=a+1;
}
else
{
    a=a+10;
}
```

```
1       LD #5
2       ST $110
3
4       MOV A, $110
5       CMP A, #8
6       JE bfalse1
7
8       MOV B, $110
9       ADD B, #1
10      ST $120
11
12      LD $120
13      ST $110
14
15      JMP if1
16
17      bfalse1:
18      MOV C, $110
19      ADD C, #10
20      ST $121
21
22      LD $121
23      ST $110
24
25  if1 NOP
26
27  return  .end
```

10. **Example 10 :**

```
if(3<=9)
{
    int c=10;
}
else
{
    int d=1;
}
```

```
1       MOV A, #3
2       CMP A, #9
3       JC btrue1
4       JNZ bfalse1
5
6       btrue1:
7       LD #10
8       ST $110
9
10      JMP if1
11
12      bfalse1:
13      LD #1
14      ST $111
15
16  if1 NOP
17
18  return  .end
```