Simple calculator

Due: 4/24

Outline

1. Introduction
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. Demo Reminder
8. More Details

Outline

1. **Introduction**
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. Demo Reminder
8. More Details

Small computer

- Let’s consider a CPU:
  - Eight 32-bit registers r0-r7. 256-byte memory.

- In this project, you need to implement a calculator.
  - **Input:** a list of **expressions**, consisting of:
    - Integers
    - Operators (+, -, *, /, =, &, |, ^, ++, --)
    - Three initial variables **x, y, z** (each having an initial value)
    - Some **new variables**
  - **Output:** A list of **assembly codes.**

- The instructions of the CPU and the clock cycles of each instruction are listed in the next page.

Outline

1. Introduction
2. **Instruction Set Architecture**
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. Demo Reminder
8. More Details

Instruction Set Architecture (I)

Opcode	Operand1	Operand2	Meaning	Cycles
MOV	Register1	Register2	Move data from register2 to register1	10
	Register1	Constant	Set the value of register1 constant	10
	Register1	[Addr2]	Move the data (4 bytes) in memory addressed Addr2 to register1. Note that Addr2 must be a multiple of 4.	200
	[Addr1]	Register2	Move the data (4 bytes) from register2 to the memory addressed Addr1. Note that Addr1 must be a multiple of 4.	200

Instruction Set Architecture (II)

Opcode	Operand1	Operand2	Meaning	Cycles
ADD	Register1	Register2	Add the values in register1 to register2 and store the result in register1	10
SUB	Register1	Register2	Subtract the value in register2 from the value in register1, and store the result in register1.	10
MUL	Register1	Register2	Multiply the values in register1 to register2 and store the result in register1	30
DIV	Register1	Register2	Divide the value in register1 by the value in register2, and store the result in register1. Note it is the integer division.	50
EXIT	Constant		Stop the program with a constant signal, whose value is specified as follows. 0: exit normally 1: the expression cannot be evaluated	20

Instruction Set Architecture (III)

Opcode	Operand1	Operand2	Meaning	Cycles
AND	Register1	Register2	Bitwise **and** the values in register1 to register2 and store the result in register1	10
OR	Register1	Register2	Bitwise **or** the values in register1 to register2 and store the result in register1	10
XOR	Register1	Register2	Bitwise **exclusive or** the values in register1 to register2 and store the result in register1	10

Outline

1. Introduction
2. Instruction Set Architecture
3. **Grammar**
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. Demo Reminder
8. More Details

A complete grammar rules

- **You should modify the package code according to the grammar below**
- statement   := END | assign_expr END
- assign_expr   := ID ASSIGN assign_expr | or_expr
- or_expr   := xor_expr or_expr_tail
- or_expr_tail   := OR xor_expr or_expr_tail | NiL
- xor_expr   := and_expr xor_expr_tail
- xor_expr_tail   := XOR and_expr xor_expr_tail | NiL
- and_expr   := addsub_expr and_expr_tail
- and_expr_tail   := AND addsub_expr and_expr_tail | NiL
- addsub_expr   := muldiv_expr addsub_expr_tail
- addsub_expr_tail := ADDSUB muldiv_expr addsub_expr_tail | NiL
- muldiv_expr   := unary_expr muldiv_expr_tail
- muldiv_expr_tail := MULDIV unary_expr muldiv_expr_tail | NiL
- unary_expr   := ADDSUB unary_expr | factor
- factor   := INT | ID | INCDEC ID | LPAREN assign_expr RPAREN

Tokens used in the grammar

- INT: integer number
- ID: variable
- ASSIGN: =
- LPAREN: (
- RPAREN:)
- END: ‘\n’
- ADDSUB: + or -
- MULDIV: * or /
- INCDEC: ++ or --
- AND: &
- OR: |
- XOR: ^

Binary Operators

- Operator priority:
  - `[*, /] > [+, -] > & > ^ > |`
- &, |, ^ are the same as **bitwise operators** in C.
- The left-hand side of an assignment (=) operator should be a **variable.**
- Valid examples:
  - `x = 5`
  - `x = y = 3`
  - `x = (y = 1)`
- Invalid examples:
  - `5 = x`
  - `x + y = 1`
  - `(x + y) = 1`

Unary Operators

- Two consecutive positive (+) or negative (-) signs should be regarded as a **prefix increment/decrement operator.** (e.g. ++x)

- Only two **consecutive (no spaces between)** signs form an increment/decrement operator, e.g.
  - **++**x should be regarded as **INC(++) x**.
  - **+ +**x should be regarded as **POSITIVE(+) POSITIVE(+) x.**

- The **operand** of an inc/dec operator should be a **variable.**
- E.g. y=++**5** or y=++**(x+6)** are both **INVALID.**

Variables (I)

- Built-in variables **x, y, z** have **initial values.**
  - Initially stored in **memory [0], [4], [8]** respectively.

- When a **NEW** variable appears, there are two cases:

1. A variable **first** appears in the **left-hand side** of an assignment operator (=):
   - **Valid,** and it can be used in the future
   - E.g. `c = x + 5`, `aa = bb = 3 * x` are both **valid.**

2. A variable **first** appears in the **right-hand side** of an assignment operator (=):
   - **Invalid,** and you should print EXIT 1
   - E.g. `x = cc * 5`, **if cc is a new variable, it is invalid.**

Variables (II)

- We **guarantee** that among all test cases, a **new variable** will **NOT** first appear in:
  - Both side of an assignment operator (=)
    - E.g. `aa = aa + 1`
  - Prefix increment/decrement expression (++ or --)
    - E.g. `x = ++bb`
- Valid variable names may contain **a-z, A-Z, numbers, and underscores(_)** and may have **arbitrary length.**
- **Name of a variable should not start with a number**
  - `Var_1` is valid, but `1_Var` is invalid
  - You should handle this error in your code
- You should **store the final values of x, y, z** in registers **r0, r1, r2** respectively before you print EXIT 0.

Outline

1. Introduction
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. **Example, Error Handling and Clock Cycles**
6. Todo
7. Demo Reminder
8. More Details

Example:

- Input: `x = z + 5`
- Output: (One of the possible outputs)

```
MOV r0, [0]
MOV r1, [4]
MOV r2, [8]
MOV r3, 5
ADD r3, r2
MOV r0, r3
EXIT 0
```

Read initial values from memory

Set a constant value 5 to r3

Add r3 (5) and r2 (z), and store the result to r3

Move the result from r3 to r0 (x)

Program ends

Error handler

- If the expression is invalid, such as

```
x = +3 % 5
y = (x+)
```

- Your final output should be

```
EXIT 1
```

- If the expression is **invalid**, no matter how many instructions you output, you should make sure your instructions contains an **EXIT 1**

Error handler - Divide by zero

- While doing division (/), if the right-hand side evaluates to zero, there are two cases:

1. **No variable** in the right-hand side:
   - `x = y / (1 + 2 - 3)`
   - This is an **invalid expression.**
2. **At least one variable** in the right-hand side:
   - ```
     x = 0
     y = 5 / x
     ```
   - This is a **valid expression.**
     ```

Total clock cycles

- Each instruction has an expected runtime:
  - Specified by **clock cycles** (in the table)
- The runtime of a program is:
  - Sum of the clock cycles of all instructions.
- Example: the following code has 90 clock cycles

```
MOV r0, 3        10 cc
MOV r1, 5        10 cc
MUL r0, r1       30 cc
MOV r1, 0        10 cc
MOV r2, 0        10 cc
EXIT 0           20 cc
```

Outline

1. Introduction
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. **Todo**
7. Demo Reminder
8. More Details

Todo – lex.h / lex.c

- Add some tokens to the TokenSet according to the complete grammar
  - The grammar is in the previous slides

- Modify the package code to accept new tokens

- Make sure your code can accept variable names with multiple characters, numbers and underscores
  - A variable starting with a number is invalid

Todo – parser.h / parser.c

- Add some parsing functions to handle with the complete grammar
- Do more error handling according to the grammar
- Handle the undefined variable error
  - The package code ignores this error now
- Make sure you deal with the divide by zero error
  - Detailed rules are in the previous slides

Todo – codeGen.h / codeGen.c

- Modify the evaluateTree() function to print assembly code
  - The provided package only calculates the answer
- Store the final value of x, y, z in registers r0, r1, r2 respectively before you print EXIT 0
- Do some optimization to reduce total cycles of the generated assembly code to get extra credits
  - You can use the provided assembly parser to calculate total cycles

Outline

1. Introduction
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. **Demo Reminder**
8. More Details

Contest

- The project has 3 parts:

1. Practice:
   - A practice contest on OJ to verify your code.
   - The practice will be available during the whole project.
2. Demo:
   - **20 test cases**, basic and advanced.
   - **Submit your code to OJ on demo day.**
3. Extra Bonus Contest:
   - Submit your code to an extra contest on demo day.
   - TAs will test the clock cycles of your generated assembly code.
   - **Less clock cycles is better.**
   - **Top 15 winners** will get extra credits.

Demo reminder

- We will create a **contest on NTHUOJ on demo day.**
  - You can send your code repeatedly within the whole day.
- There will be **20 testcases.**
  - Each valued 5 points, totally 100.
- **Also, remember to submit the same code to iLMS.**
- We will use a parser to check the correctness.
- If you have any question about the project, feel free to ask your question on iLMS or email us.

Plagiarism

- Definition
  - Copying someone else's work
  - Including variable renaming!
- Policy
  - Perfectly fine to ask questions and discuss.
  - NOT acceptable to copy part or whole work.
    => zero points on the project
  - **Automatic tools will be used to catch cheaters.**
- TAs of the two classes will work together to eliminate plagiarisms. **Do not cheat!!** ☺

Outline

1. Introduction
2. Instruction Set Architecture
3. Grammar
4. Operators and Variables
5. Example, Error Handling and Clock Cycles
6. Todo
7. Demo Reminder
8. More Details

Test case restriction

- Only specified variables and signs are allowed.
  - **Allowed:** + - * / ( ) & | ^ 0-9 a-z A-Z _
  - **Not allowed:** @ # $ % … and more
  - Symbols not allowed will not appear in the test cases.
- The result should follow the elementary arithmetic.
  - `x=2+3*4`, x will be 14
  - `x=(2+3)*4`, x will be 20
- A test case may contain multiple expressions
- Multiple expressions are separated by “\n”.

Program output restrictions

- You should **print all the assembly code.**
- Don’t just print the result of x, y, z.
- Ex:

```
y = 1
z = 0
x = y + 5
```

- One of the possible correct result will be:

```
MOV r0, 0
MOV r1, 1
MOV r2, 0
ADD r0, r1
MOV r3, 5
ADD r0, r3
EXIT 0
```

Others

- You must deal with basic errors using EXIT 1.
  - Extra or missing symbols (e.g. `x = &|3` or `x = y 3`)
  - Incorrect expression (e.g. `1 = 2 + 3` or `++y = 5`)
- You must deal with the uninitialized variable error
  - Variables should appear in the left hand side of = first
  - The provided package now ignores this error
- Should be able to accept variables with arbitrary length
  - E.g. `var_1, tmpVal2 are both valid names`
- Should be able to calculate numbers > 10.
  - E.g. `x = 11`
- Should be able to calculate long expressions.
  - E.g. `x = 4-x/z*33+4+20*31+10`