# SWAGASM'07:
# SIMD Within A Gate Array Assembler

November 25, 2007

This document summarizes the assembler project.

## 1 SWAGASM'07 AIK Specification

Although you are to write your assembler from scratch (well, reusing only code that you authored for earlier projects), one of the most concise descriptions of what you are to do is the following fully functional AIK specification:

```
.noarg          := .this:8
.alias .noarg { AND OR XOR SRL RIGHT LEFT ADD SUB LT GT
POP RET DISABLEZ PUSHEN POPEN RESET ALL INPUT OUTPUT }
LOAD @ .x       := 0x2:4 .x:4
LOAD $ .x       := 0x3:4 .x:4
STORE @ .x      := 0x4:4 .x:4
STORE $ .x      := 0x5:4 .x:4
PUSH .x ?((.x<=7)&&(.x>=-8)) := 0x6:4 .x:4
PUSH .x         := 0x7:4 0:4 .x:32
JNONE .x ?(((.x-.)<=7)&&((.x-.)>=-8)) := 0x8:4 (.x-.):4
JNONE .x        := 0x9:4 0:4 .x:16
JUMP .x ?(((.x-.)<=7)&&((.x-.)>=-8)) := 0xa:4 (.x-.):4
JUMP .x         := 0xb:4 0:4 .x:16
CALL .x         := 0xc:4 0:4 .x:16
.const 8 NPROC
.alias .equate EQU
.alias .set SET
.alias .if IF
.alias .end ENDIF
.segment TEXT 8 0x10000 0 .VMEM
```

Using this specification with the Assembler Interpreter from Kentucky:

```
http://aggregate.org/AIK
```

yields an assembler that is fully compatible with the one you're building from scratch. It even generates its output as a Verilog MEMory (`.VMEM`) image file.

## 2 SWAGASM'07 Language Syntax

SWAGASM'07 is a fairly typical assembler. The following extended BNF grammar, combined with the same lexical conventions used by your compiler project, define the language structure.

```
prog: (stat)* EOF
    ;
stat: WORD labstat
    | “AND”
    | “OR”
    | “XOR”
    | “SRL”
    | “RIGHT”
    | “LEFT”
    | “ADD”
    | “SUB”
    | “LT”
    | “GT”
    | “PUSH” expr
    | “POP”
    | “JNONE” expr
    | “JUMP” expr
    | “CALL” expr
    | “RET”
    | “DISABLEZ”
    | “PUSHEN”
    | “POPEN”
    | “RESET”
    | “ALL”
    | (“LOAD” | “STORE”) ('@' | '$') expr
    | “INPUT”
    | “OUTPUT”
    | “IF” expr
    | “ENDIF”
    ;
labstat: ':' stat
       | (“EQU” | “SET”) expr
       ;
expr: expr1 (('&' | '|' | '^') expr1)*
    ;
expr1: expr2 (('<' | '>') expr2)*
     ;
expr2: expr3 (('+' | '-') expr3)*
     ;
expr3: expr4 ('*' expr4)*
```

```
      ;
expr4: NUMBER
      | WORD
      | “NPROC”
      | '-' expr4
      | '!' expr4
      | '~' expr4
      | '(' expr ')'
      ;
```

There are two features of this assembly language that require special treatment:

- There are three potentially span-dependent instructions: `PUSH`, `JNONE`, and `JUMP`. Each has a short form with a signed 4-bit value and a long form which has a 32-bit value for `PUSH` and 16-bit values for `JNONE` and `JUMP`. Note that the value in a `PUSH` is the value of the expression, whereas the other SDIs encode an offset from the current location. Multiple passes will be needed to resolve these SDIs.
- The nestable `IF` construct works very much like the C preprocessor's `#if`, conditionally “commenting out” the code between an `IF` and the corresponding `ENDIF`. Note that what was commented out in one pass might not be commented out in another pass....

Your lexical analyzer was designed to read from `stdin`, which generally cannot be “rewound” to make multiple passes. Thus, you will have to modify your lexical analyzer. This can be done trivially by reading the entire input into an array (you may assume the input is no more than 1024*1024 characters) and then replacing `getchar()` with something like:

```
#define getchar() inputbuffer[++inputpos]
```

or whatever names you select for your input buffer and current position. Rewinding the input is simply setting the input position to 0. Note that declaring the input buffer as holding `short`, rather than `char`, values neatly eliminates the potential problem with EOF returning `-1`.

## 3 Instruction Set Encoding

Perhaps the clearest explanation of the encoding is the AIK description given above; it is also online at `http://aggregate.org/CS/aik.html` .

| Instruction | Arguments | Encoding |
|---|---|---|
| `AND` | | 0x00 |
| `OR` | | 0x01 |
| `XOR` | | 0x02 |
| `SRL` | | 0x03 |
| `RIGHT` | | 0x04 |
| `LEFT` | | 0x05 |
| `ADD` | | 0x06 |
| `SUB` | | 0x07 |
| `LT` | | 0x08 |
| `GT` | | 0x09 |
| `PUSH` | v | 0x6? or 0x70 0x???????? |
| `POP` | | 0x0a |
| `JNONE (BNONE)` | a | 0x6? or 0x70 0x???? |
| `JUMP (BRANCH)` | a | 0x8? or 0x90 0x???? |
| `CALL` | a | 0xc0 0x???? |
| `RET` | | 0x0b |
| `DISABLEZ` | | 0x0c |
| `PUSHEN` | | 0x0d |
| `POPEN` | | 0x0e |
| `RESET` | | 0x0f |
| `ALL` | | 0x10 |
| `LOAD` | `@` place | 0x2? |
| `LOAD` | `$` place | 0x3? |
| `STORE` | `@` place | 0x4? |
| `STORE` | `$` place | 0x5? |
| `INPUT` | | 0x11 |
| `OUTPUT` | | 0x12 |

Table 1: SWAGA Instruction Set Summary

## 4 Output File Format

What you will generate is a Verilog MEMory (`.VMEM`) image file. It is a very simple format that looks like:

```
//generated by ... comment
@1234
42
ab
//end
```

Here, the `@0000` line sets the origin at `0x1234`. Each subsequent line holds the value of one byte, with a low-byte-first byte order. In other words, the above encodes the 16-bit value `0xab42`.

## 5 Due Date & Submission Requirements

This project is due **November 30, 2007**. It should be submitted via the the appropriate WWW form on the course site as a "tarball" created using `tar -c`. It must contain at least three files:

1. Your C source code, in one or more files.
2. A `Makefile` which will build an executable file called `swagasm` by simply typing `make`.
3. Minimal "implementor's notes" describing how your code works... or fails to work. The document should be formatted as very straightforward HTML and placed in a file named `notes.html`. The HTML should be self-contained, not referencing anything outside of the contents of the tarball. Code which is broken but documented as such will get only half as many points taken off for such errors. This is true even to the extent of an empty assignment being given half credit if the documentation says something like "This doesn't work because I didn't get around to writing the project code."