SWAGAC'07: SIMD Within A Gate Array C Compiler

November 20, 2007

This document summarizes the last phase of the first compiler project: construction of the code-generation phase for SWAGAC'07, the SIMD Within A Gate Array C dialect. Generating code is largely a matter of understanding the target machine, SWAGA'07.

The machine you will be generating code for is a simple attached processor designed to be implemented in an FPGA as a special-purpose accelerator. It is not capable of doing general computing, but instead is entirely focussed on extending the processor with an explicitly fed pipeline. The main processor would send a data stream to a FIFO buffering inout to the SWAGA processor and would read a result data stream from another FIFO buffering output; the SWAGA processor has no other I/O mechanisms. Similarly, SWAGA has no protection hardware or other operating system support; it is expected that the host OS would ensure that only one host process at a time was given access to the SWAGA hardware.

Aside from adding code to your parser to generate assembly code to `stdout`, you will need to modify your symbol table again... to keep track of the location/offset for each variable. Oddly enough, your compiler really does not need to distinguish between `mono` and `poly` stuff because the hardware really treats everything as `poly`.

1 Assembly Language & Instruction Set

The SWAGA'07 instruction set is essentially an integer SIMD stack instruction set with a few quirks. The quirks are about evenly split between making your life easier and making the hardware more efficient. All be warned that, despite many similarities, this is *not* the same instruction set used for SWAGA last year.

Instruction	Arguments	Function (as C code)
AND		push(pop() & pop())
OR		push(pop() | pop())
XOR		push(pop() ^ pop())
SRL		push(pop() >>= 1)
RIGHT		push(PE[(IPROC-1)%NPROC].pop())
LEFT		push(PE[(IPROC+1)%NPROC].pop())
ADD		push(pop() + pop())
SUB		a=pop(); push(pop() - a)
LT		a=pop(); push(pop() < a)
GT		a=pop(); push(pop() > a)
PUSH	value	push(value)
POP		pop()
JNONE (BNONE)	addr	if (all_disabled) PC = addr
JUMP (BRANCH)	addr	PC = addr
CALL	addr	push_return_addr(); PC = addr
RET		PC = pop_return_addr();
DISABLEZ		disable_where(pop() == 0)
PUSHEN		save_enable_state()
POPEN		restore_enable_state()
RESET		initialize everything
ALL		enable_all_PEs()
LOAD	place	push(STACK[place])
STORE	place	push(STACK[place]=pop())
INPUT		push(get(input_fifo))
OUTPUT		put(output_fifo, pop())

Table 1: SWAGA Instruction Set Summary

The assembly language also provides a few other nice features. The same kind of expressions used in SWAGAC'07 are also allowed for constant values and addresses in assembly language instructions. There also are assembly-time constants (by `EQU`) and variables (by `SET`). For example, all the following are ways of saying `PUSH 42`:

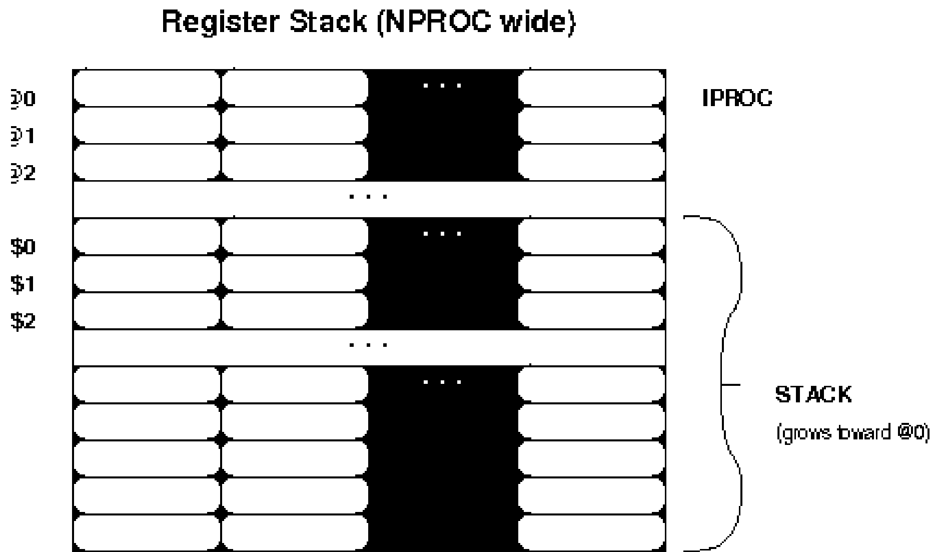
```
a EQU 40
b SET a
  PUSH 42
  PUSH c
```

```
  PUSH b+2
b SET 42
  PUSH 2+a
c EQU 84-b
  PUSH b
```

Also note that execution begins with the statement labeled `start`, not main. This is significant because the main function can be recursive, so you cannot put system initialization code after the label at the beginning of the main function.

1.1 Processor Model

The SWAGA instruction set is a very simple design generally following RISC principles, but using a register-stack addressing scheme to simplify the instructions. Further, there is no data memory other than this register file. The register file is a fixed size, assumed to be at least 64 lines long. It looks like:



The `LOAD` and `STORE` instructions are perhaps the most strange in that they provide two forms of addressing for the register file, as illustrated in the figure above. The `@` addressing refers to a fixed set of registers, whereas the `$` addressing refers to registers relative to the top of the stack.

The only way to get a constant into the system is to `PUSH` a scalar value. There is, however, one constant that the compiler does not know the value of: `NPROC`. This value should be kept symbolic in your assembly language output and the assembler will make the appropriate substitution. Note that `IPROC` is not a constant, but a vector of constants. The hardware keeps `IPROC` in `@0`.

There are two additional hardware stacks, an `NPROC`-bit-wide SIMD enable stack and a scalar return address stack. The enable stack is manipulated by

the PUSHEN, POPEN, DISABLEZ, and ALL instructions. DISABLEZ and ALL merely impact the current (top-of-stack) enable state, they do not push nor pop. The return address stack is accessed only through CALL and RET.

How does one initialize the three stacks? Well, that's what RESET is for. After RESET, the register stack is empty, the enable stack has only one entry (in which all processing elements are enabled), and the return stack is empty (so that a RET will halt the processor).

The last oddity is the handling of conditionals. There is an explicit enable mechanism that not only serves to selectively disable processing elements, but also is used as a type of parallel condition code register. JNONE, the only conditional control flow construct, alters control flow only if no processing elements are enabled. Notice that JNONE and JUMP are jumps, not branches, so there is no restriction on the location of the target. However, there actually are also branch forms, BNONE and BRANCH, which the assembler automatically will substitute for JNONE and JUMP whenever the spans allow.

2 Example Code

A live version of the compiler will be available at the course WWW site to help you understand what code to generate. However, the following example code should guide your translation process.

```
mono    one;
poly    two;
twe(mono x, poly y){
        if (y < two) {
                two = IPROC;
                where (x < y) {
                        two = NPROC;
                }
                if (x < one) {
                        y = two = two + 1;
                }
        }
        while (one < two) {
                two = two - y;
        }
        return(y & 5);
}
main(mono arg)
{
        two = twe(42, input);
        output two;
}
```

The assembly code output should work like (but not necessarily be identical to) the following. Note that this instruction set has been dramatically simplified, so it is highly likely that multpile students will generate code that differs only in cosmetic ways such as comments, use of `EQU` and `SET` for symbolic names, and names of automatically generated labels.

```
;       compiled by swagac07 version 20071109
start:
        RESET
        JUMP    main
;       start of function twe
twe:
;       if statement
        PUSHEN
        LOAD    $0
        LOAD    @2
        LT
        LOAD    $0
        DISABLEZ
        JNONE   _0
;       expression statement
        LOAD    @0
        STORE   @2
        POP
;       where statement
        PUSHEN
        LOAD    $2
        LOAD    $2
        LT
        LOAD    $0
        DISABLEZ
;       expression statement
        PUSH    NPROC
        STORE   @2
        POP
        POPEN
        POP
;       if statement
        PUSHEN
        LOAD    $2
        LOAD    @1
        LT
        LOAD    $0
        DISABLEZ
        JNONE   _1
;       expression statement
```

```
        LOAD    @2
        PUSH    1
        ADD
        STORE   @2
        STORE   $3
        POP
_1:
        POPEN
        POP
_0:
        POPEN
        POP
;       while statement
        PUSHEN
_2:
        LOAD    @1
        LOAD    @2
        LT
        DISABLEZ
        JNONE   _3
;       expression statement
        LOAD    @2
        LOAD    $1
        SUB
        STORE   @2
        POP
        JUMP    _2
_3:
        POPEN
;       return statement
        LOAD    $0
        PUSH    5
        AND
        STORE   $3
        POP
        POP
        POP
        RET
;       default return
        POP
        POP
        RET
;       end of function
;       start of function main
main:
;       expression statement
```

```
        PUSH    0
        PUSH    42
        INPUT
        CALL    twe
        STORE   @2
        POP
;       expression statement
        LOAD    @2
        OUTPUT
        POP
;       default return
        POP
        RET
;       end of function
```

3 Due Date & Submission Requirements

This project is due **November 9, 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 `swagac` 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."