SIMDC12 Compiler

March 20, 2012

Now that you have all completed your parsers (What! You haven't! Well do it now!), it's time to generate some code....

1 The Target Instruction Set

The target instruction set for your compiler is the very simple general-register design described here:

Instruction	Arguments	Function (as C code)
`AND`	rd, rs, rt	rd = (rs & rt)
`OR`	rd, rs, rt	rd = (rs | rt)
`XOR`	rd, rs, rt	rd = (rs ^ rt)
`LT`	rd, rs, rt	rd = (rs < rt)
`CONST`	rd, c	rd = c
`MUL`	rd, rs, rt	rd = (rs + rt)
`ADD`	rd, rs, rt	rd = (rs + rt)
`NEG`	rd, rs	rd = (-rs)
`NOT`	rd, rs	rd = (~rs)
`LNOT`	rd, rs	rd = (!rs)
`LD`	rd, rs	rd = mem[rs]
`ST`	rd, rs	mem[rs] = rd
`PUT`	`@`rd, rt, rs	rd.rt = rs
`GET`	`@`rd, `@`rs, rt	rd = rs.rt
`GOR`	`$`rd, `@`rs	rd = globalOR(enabled rs)
`JZ`	rs, `$`rd	if (any(rs==0)) PC = rd
`PUSH`	rd	mem[SP–] = rd
`POP`	rd	rd = mem[++SP]
`DZ`	rd	if (enable & !rd) disable
`SEN`		save enable state
`REN`		restore enable state

Table 1: SIMDC12 Instruction Set Summary

Each of these instructions is fairly straightforward in its operation, except in that the target machine is a SIMD with the ability to perform operations on both the PEs and the CU. The instruction set is closely related to that of the MIPS architecture used in EE380/CS380. Registers named using $ reside in the CU, whereas those named with @ are replicated in each PE and may have a different value in each. The 16 registers in each are:

Register	Mono (`$`)	Poly (`@`)
`0`	Always 0	Always 0
`1`	NPROC	IPROC
`2`	$SP	@SP
`3`	Return Value	Return Value
`4-15`	General Registers	General Registers

Table 2: SIMDC12 Registers

Generating code is fairly straightforward, and is made simpler by the fact that you may assume that your compiler never runs out of registers for local expression evaluation. You also should note that, unlike MIPS, there are stack PUSH and POP instructions that make it easy to use the stack for argument passing (and for register spill/reload, although you need not worry about that here).

Perhaps the most confusing aspect is the conversion between mono and poly. The CU fetches, decodes, and broadcasts the relevant control for each instruction. Thus, a PE instruction can apparently read values not only from @ registers, but also from $ registers. The @ references are broadcast as register references, whereas the CU broadcasts the "constant" value taken from its $ register when a PE references one. However, only the CU can write to a $ register, so the type of the destination register really distinguishes CU and PE instructions. For example, `ADD $4,$5,$6` is a CU instruction, but `ADD @4,$5,$6` is executed by each PE. The only way for the CU to get data from one of more PEs is the "global OR", or `GOR`, instruction. For example, `GOR $4,@5` will bitwise-OR the values from register 5 in all enabled PEs, placing the result in CU register 4. There is an even messier issue involving transfer of enable state for the "." communication construct and the `PUT` and `GET` instructions... however, it is so messy that we're going to ignore that construct for now.

Control flow is also a bit strange... if fairly ordinary for a SIMD. PEs have hardware enable masking, with the enable state controlled by `SEN`, `REN`, and `DZ`. Note that enable masking is entirely structured; there is no way to directly change from one enable state to another. The CU has much more conventional (unstructured) control in the form of a "jump zero" instruction, `JZ`. The odd thing is, that's the only control flow instruction it has. An unconditional jump to label `abc` would use a sequence like `CONST $4,abc` followed by `JZ $0,$4`. The same sequence is used for calling a subroutine, except the return address has to be pushed too. Given that, it is should not be too surprising that a return from a subroutine is implemented by `JZ` preceeded by getting the return address from the stack. A conditional jump simply uses a register other than `$0` as the first operand in the `JZ`. Note that `JZ` also can test if any register in an enabled PE is 0. Darned useful instruction, that `JZ`.

Did I just mention calls? Well, we need to agree on how those are done. The answer is very simple: everything but the return value is on the stack. The correct response to that is "which stack?" The answer is all of them. The mono stack should be used for holding mono values, the most obvious of which is the return address. The poly stacks should be used for poly values... and disabled PEs don't necessarily have the same stack pointer values as enabled ones. Aside from the split between mono and poly stacks, the frame is simply pushed in the order of: arguments (evaluated in left-to-right order), then return address, then any locals. The called function is expected to remove the return address (and everything above that) from the stack before returning; the caller is expected to remove the argument values. Any registers numbered 4 or higher that you use in a function should have their values saved on the stack inside the function and restored at the end of the function. It is acceptable for your compiler to save and restore registers it didn't use... e.g., all 24 of them. Keep in mind there is no frame pointer, so you need to keep track of how much is on the stack. For that, you also need to know that `$SP` and `@SP` stacks grow downward and always point at the first unused word on the stack (i.e., 1 address below the datum on "top" of the stack). Actually, if you read the descriptions of `PUSH` and `POP` carefully, you knew this.

2 Assembler Pseudo-Operations/Directives

In addition to the instructions, there are some assembler directives you need to know:

Pseudo-Instruction	Function
`MONO`	mono data segment follows
`POLY`	poly data segment follows
`CODE`	(mono) code follows
`name:`	label name gets this address (forever)
`name EQU value`	label name gets this value (forever)
`name SET value`	label name gets this value (forward)
`WORD value`	allocate one word with value

Table 3: SIMDC12 Instruction Set Summary

The three segments should not be a surprise, and you were probably expecting WORD. The thing that often throws people is the difference between EQU and SET. Normally, assembler labels have values that are carried forward and backward in the source code; a label can be defined lexically after it is used. They are essentially assembly-time constants. In contrast, SET is an assembly-time variable assignment, the value of which is only seen lexically after each (of possibly many) definitions. Thus, SET is not really needed, but is a convenience feature. It is most often used for things like naming the stack offset to a local variable.

3 Tuple Optimizations

Well, you didn't think I was going to give you such a simple instruction set and have that be all, did you? You're going to do the you-should-be-embarrassed-not-to-do-them standard basic block optimizations.

We have talked about the close relationship between pretending to generate stack code, building trees, and building tuples. Well, I like to think of building things by calling functions that do the clever stuff and return the appropriate tuple pointer or index of the result. These functions do need to scan the current block's tuples. Of course, they only need to do that when optimizing – **initially, you should implement all this as just making a new tuple every time, and then add the optimization after you know the basic stuff works.**

```
tuple doop(opcode o, tuple t0, tuple t1)
{
  /* AND, OR, XOR, LT, MUL, ADD, NEG, NOT, LNOT, GOR */
  if (o is commutative) {normalize t0,t1 order}
  if (local optimization applies) {do it; return result tuple}
  if (both t0 and t1 constant) {return doc(folded result)}
  if (equivalent tuple available) {return that}
  make new tuple for o(t0, t1);
  mark tuple as non-constant-valued;
  return(new tuple);
}
tuple doc(int c)
{
  /* CONST */
  if (equivalent tuple available) {return that}
  make new tuple for const(c);
  mark tuple as constant-valued;
  return(new tuple);
}
tuple dold(tuple t)
{
  /* LD */
  if (value of memory[t] is known) {return that}
```

```
  make new tuple for ld(t);
  mark tuple as non-constant-valued;
  return(new tuple);
}
tuple dost(tuple t0, tuple t1)
{
  /* ST t0, t1 */
  make unavailable all ambiguous aliases of memory[t1];
  make a new tuple for st(t0, t1);
  return(new tuple);
}
```

Other instruction types do the obvious things. Note, however, that the special registers need special treatment throughout.

I generally find it easiest to pre-allocate dummy tuples in every block for special things. Obviously, the constant 0, which is `$0`, deserves such a dummy tuple. The same is true of `NPROC` and `IPROC`. The `SP` is a little strange (a very special case), but to access arguments and local variables on the stack, you'll need to have tuples to reference for both the mono and poly versions. The return value register does not need special modeling if you simply make the result of a return statement get copied into it.

Notice that throughout all the above you will need to be careful to distinguish between mono and poly values....

Oh yes. What are the local optimizations mentioned above? Minimally:

Initial Operation	Optimized Result
`or(t, const(0))`	`t`
`xor(t, const(0))`	`t`
`mul(t, const(0))`	`const(0)`
`add(t, const(0))`	`t`
`neg(neg(t))`	`t`
`not(not(t))`	`t`
`or(t0, t1) where t0=t1`	`t0`
`xor(t0, t1) where t0=t1`	`const(0)`

Table 4: Local Optimizations

It goes without saying (but I'll say it anyway) that your compiler should not spit-out anything for a basic block until the tuples have been processed. Buffering beyond that level is not necessary. Similary, you are free to flip between segments, so mono and poly global data intermized with code can be output in the sequence it was recognized, provided each is put in the correct segment by the `MONO`, `POLY`, and `CODE` directives.