Problem Set 4 Solutions

Problem 1 [100]: A PostFix to 6811 Translator

You were asked to write a PostFix to 6811 translator from scratch. Here we present one possible solution in stages.

A Code Abstraction

To abstract over the process of code generation, it is helpful to have an abstraction for gluing code fragments together. Such an abstraction makes it easy to experiment with different strategies, such as storing compiled subroutines “in-line” vs. storing compiled subroutines in a different part of memory. Fig. 1 presents the signature for a code abstraction we will use. Code fragments are created by \texttt{gen}, \texttt{genStr}, and \texttt{genSubr} functions; they are glued together by \texttt{glue}, and they are converted into 6811 instructions by \texttt{seq}.

\begin{verbatim}
  type code = The abstract type of the code fragments being generated.
  val gen: Instr68HC11.instr list -> code
         |Turn a “main” 6811 code sequence into a code fragment.
  val genSubr: code -> (Instr68HC11.label -> code) -> code
            |Given the code fragment c for a subroutine body and a function f that maps a label to code, returns
            |a code fragment that abstractly contains 2 parts:
            |1. the code c labeled with new label l.
            |2. The code that results from invoking f on l.
  val genStr: string -> (Instr68HC11.label -> code) -> code
            |Given a string s and a function f that maps a label to code, returns a code fragment that abstractly
            |contains 2 parts:
            |1. A sequence of bytes labeled with a new label l that begins with a length byte and is followed
               by the characters of s. It is an error if s has more than 255 characters.
            |2. The code that results from invoking f on l.
  val glue: code list -> code
            |Returns the result of gluing together a list of code fragments.
  val seq: code -> Instr68HC11.instr list
            |Converts a code fragment into executable 6811 code.
\end{verbatim}

Figure 1: Signature of the code abstraction.

The simplest implementation of the code abstraction is shown in Fig. 2. Here the \texttt{code} type is simply a list of 6811 instructions. Both \texttt{genSubr} and \texttt{genStr} are implemented in terms of a \texttt{gen2} function that has its first (i.e., main) instruction sequence “jump around” the second instruction sequence. It is assumed that the second instruction sequence could be anywhere in memory; it just happens to be plunked down in the middle of the main instruction sequence. We assume the existence of a \texttt{newLabel} function that generates fresh labels. Here is one such function:

\begin{verbatim}
  let newLabel = StringUtils.genFresh "_"
\end{verbatim}
A fancier implementation of the code abstraction is shown in Fig. 3. Here the code type is a triple of 6811 instruction lists: (1) the main code; (2) subroutine code; and (3) string constants. In this representation, the glue function performs an elementwise concatenation of the sequence components of a list of triples. The gen function adds code to the first component, genSubr c f glues the code that results from calling f together with subroutine code c in the subroutine component, and genStr s f glues the code that results from calling f together with labeled string bytes in the string component.

One last utility we will need is a way of generating temporary addresses:

\[
\text{tempAddress}(i, \text{temp}) = \text{AddAddr}(\text{AddrLabel} \text{"temps"}, \text{Addr}(\text{temp} + i))
\]

The Main Compiler

The main compiler code is presented in Fig. 4. The code begin code precedes the compiled body of the given program. Note how that body is given the label postfix-code and is treated as a subroutine call. This simplifies the handling of tail recursion, since every command sequence compiled by compComs can be assumed to end in an rts in a naive compilation.

In Fig. 4, assembly code string stands for a string defining many assembly code subroutines. These are presented in Figs. 5–8. These subroutines are a good way to avoid including common code idioms in the compiled body code. In some cases, these subroutines are not as efficient as they could be. For instance, Ivana and Mirena had much cleverer versions of need1, need2, and
let gen instrs = (instrs, [], []) (* Generate main code *)

let genSubr subrCode fcn =  
  let subrBegin = newLabel "subr_begin"  
  and (mains, subrs, strs) = subrCode  
  in glue [fcn subrBegin;  
           ([], [Label subrBegin] @ mains @ subrs, strs)]

let genStr str fcn =  
  let strBegin = newLabel "string_begin"  
  in glue [fcn strBegin;  
            ([], [], [Label strBegin; FormJavaString str])]

let glue triples =  
  let (mainss, subrss, strss) = ListUtils.unzip3 triples  
  in List.concat mainss, List.concat subrss, List.concat strss

let seq (mains, subrs, strs) =  
  mains @ subrs @ strs  
  @ [FormBytes [0;0;0]; (* Add a few extra byte to protect code in lieu of alignment *)  
     Label "pfstkmax"] (* Max range of PostFix stack *)

Figure 3: A fancy representation of code as a triple of instruction lists: (1) main code (2) strings and (3) subroutines.

need3 that used address arithmetic rather than inx and dex.
let rec compile (Pgm(n,coms)) = 
    seq (glue [beginCode n; compComs coms])

    and beginCode n = (* Standard boilerplate for beginning of program *)
        glue
        [gen
            (Parser.parseString assembly code string);
            glue (List.map
                (fun i ->
                    gen
                    [ Label ("read-arg" ^ (string_of_int i));
                        Ldd (Imm (ImmLabel ("arg" ^ (string_of_int i) ^ "-string")));
                        Std (Ext (AddrLabel "prompt"));
                        Js r (Ext (AddrLabel "read-signed-word"));
                        Ldd (Ext (AddrLabel "wordread"));
                        Jsr (Ext (AddrLabel "pushd"));
                        Rts;
                    Label ("arg" ^ (string_of_int i) ^ "-string");
                        FormJavaString ("arg" ^ (string_of_int i))
                    ]
                )
            )
            (ListUtils.fromTo 1 n));
        gen (Parser.parseString
            "main-loop:
            lds #$ffff ; Reset control stack ptr
            ldx #pfstkmin ; Store initial stack ptr in X
            ");
        glue (List.map (fun i ->
            gen [Jsr (Ext (AddrLabel ("read-arg" ^ (string_of_int i)))))
        )
            (ListUtils.fromTo 1 n));
        gen (Parser.parseString
            "
            jsrlcd-clear
            jsr postfix-code ; Invoke subroutine for body of PostFix program
            jsr popd
            jsr lcd-bottom
            ldx #result-string
            jsr display-string
            jsr display-equal
            jsr display-signed-word
            jsr wait-for-stop
            jmp main-loop ; Evaluate again when stop pressed
            ");
        glue (Parser.parseString
            "postfix-code: ; Code for body of PostFix program goes here
            ")
    ]

Figure 4: The main compiler code. The part labeled assembly code string is presented later.
include ../hc11/prolog.asm ; Include standard libraries
pfstkmin equ $bf00 ; Bottom of PostFix stack
main: ; Start of user program
  lds #$ffff ; Put stack at end of memory
  jsr init ; Perform initializations
  jmp main-loop ; Start main loop
div0-error:
  jsr lcd-clear
  ldx #div0-string
  jsr display-string
  jsr display-signed-word
  jsr wait-for-stop
  jmp main-loop ; Evaluate again when stop pressed
rem0-error:
  jsr lcd-clear
  ldx #rem0-string
  jsr display-string
  jsr display-signed-word
  jsr wait-for-stop
  jmp main-loop ; Evaluate again when stop pressed
put-neg-index:
  jsr lcd-clear
  ldx #put-neg-string
  jsr display-string
  jsr display-signed-word
  jsr wait-for-stop
  jmp main-loop
get-neg-index:
  jsr lcd-clear
  ldx #get-neg-string
  jsr display-string
  jsr display-signed-word
  jsr wait-for-stop
  jmp main-loop
pfstkfull:
  jsr lcd-clear
  ldx #full-string
  jsr display-string
  jsr wait-for-stop
  jmp main-loop
cstkfull:
  jsr lcd-clear
  ldx #cfull-string
  jsr display-string
  jsr wait-for-stop
  jmp main-loop
Figure 5: Assembly Code, Part 1
Compiling Command Sequences

The function \texttt{compComs} for compiling command sequences is shown below:

\begin{verbatim}
(* (compComs exp) compiles command list coms into a sequence of 6811 instructions that has the effect of changing the run-time in the way that executing the commands would *)
and compComs coms =
match coms with
| [] -> gen [Rts]
| [Exec] -> gen [Jmp (Ext (AddrLabel "exec"))]
(* Tail call for exec handled specially! *)
| (c::cs) -> glue [compCom c; compComs cs]
\end{verbatim}

It is assumed that every command sequence ends in a return; the top level program must be embedded in a subroutine for this to make sense. Tail recursion is implemented by handling a sequence ending in \texttt{exec} as a \texttt{jmp} rather than as a \texttt{jsr} followed by an \texttt{rts}.

This is not the only way to handle tail recursion. An alternative is to first generate \texttt{jsr/rts} pairs, and then remove them by a separate peephole optimization pass.
Figure 7: Assembly Code, Part 3
exec:
    sts 0,X ; Test control stack pointer
    ldy #$c100 ; to make sure it's not too close to reset vector.
    cpy 0,X
    bhi cstkfull
    jsr need1
    ldy 0,X
    jmp 0,Y ; Beware: Jsr jumps to effective address,
    ; not *contents* of effective address.
    ; tail optimization
result-string:
    fjs result"
div0-string:
    fjs Div by 0:"
get-neg-string:
    fjs Get index:" put-neg-string:
    fjs Put index:" rem0-string:
    fjs Rem by 0:" empty-string:
    fjs Empty pstack" full-string:
    fjs Full pstack" cfull-string:
    fjs Full cstack"
";

Figure 8: Assembly Code, Part 4
Compiling Commands

The core of the compiler is the code generation for individual PostFix commands. One approach is shown in Figs. 9–13. These code generators were designed to emphasize the efficiency of the generated code over the readability of the compiler. For instance, \texttt{swap} could be compiled via a sequence of calls to pop and push subroutines, but the generated code given here is much more efficient. The handling of \texttt{mul} in Fig. 12 is particularly tricky. In Fig. 13, note how \texttt{compRel} abstracts over the common pattern of compiling relational operations.

\begin{verbatim}
(* (compCom exp) compiles command com into a sequence of 6811
 instructions that has the effect of changing the run-time in the
 way that executing the command would *)
and compCom com =
  match com with
  | Int i -> glue [compInt i; gen [Jsr (Ext (AddrLabel "pushd"))]]
  | Str s -> genStr s
    (fun strLabel ->
      gen [Ldd (Imm (ImmLabel strLabel));
        Jsr (Ext (AddrLabel "pushd"))]) (* Push string ptr on pfstk *)
  | Seq coms ->
    genSubr (compComs coms)
      (fun subrLabel ->
        gen [Ldd (Imm (ImmLabel subrLabel));
          Jsr (Ext (AddrLabel "pushd"))]) (* Push subroutine ptr on pfstk *)
  | Exec -> gen [Jsr (Ext (AddrLabel "exec"))] (* Non-tail call for exec *)
  | Pop -> gen [Jsr (Ext (AddrLabel "popd"))]
  | Swap ->
    gen [Jsr (Ext (AddrLabel "need2"));
      Ldd (Index(0,X)); Xgdy;
      Ldd (Index(2,X)); Xgdy;
      Std (Index(2,X)); Xgdy;
      Std (Index(0,X)); Dex;
      Dex]
  | Prs ->
    gen [(* Jsr (Ext (AddrLabel "popd"));
        Pshx;
        Xgdx; *)
      Jsr (Ext (AddrLabel "need1"));
      Pshx;
      Ldx (Index(0,X));
      Jsr (Ext (AddrLabel "display-string"));
      Jsr (Ext (AddrLabel "wait-for-start-stop"));
      (* Jsr (Ext (AddrLabel "wait-1msec")); (* Give it time to display *) *)
      Pulx]
  | Pri -> gen
    [Jsr (Ext (AddrLabel "popd"));
     Jsr (Ext (AddrLabel "display-signed-word"));
     (* Jsr (Ext (AddrLabel "wait-1msec")); (* Give it time to display *) *)
     Jsr (Ext (AddrLabel "wait-for-start-stop"))]
\end{verbatim}

Figure 9: Compiling PostFix commands, part 1.
| Sel -> |
let selTrue = newLabel "sel-true"
and selJoin = newLabel "sel-join"
in gen

[Jsrb (Ext (AddrLabel "need3")); (* Ensure we at least have an index; leaves PFSP pointing at alternate *)
Ldb (Index(4,X)); (* Load test into D *)
Bne (BraLabel selTrue);
Ldd (Index(0,X)); (* Execute this if test = 0; i.e., false case *)
Bra (BraLabel selJoin);
Label selTrue;
Ldd (Index(2,X)); (* Execute this if test != 0; i.e., true case *)
Label selJoin; (* Join point for two branches *)
Std (Index(4,X)); (* Store chosen val into former test slot *)
Inx; (* and pop consequent *)
Inx;]

| Get -> |
let nonNegLabel = newLabel "get-non-neg"
in gen

[Jsrb (Ext (AddrLabel "need1")); (* Ensure we at least have an index; leaves PFSP pointing at index *)
Ldd (Index(0,X)); (* Load index into D *)
Bge (BraLabel nonNegLabel); (* Test for negative index *)
Jmp (Ext (AddrLabel "get-neg-index")); (* Report error for negative index *)
Label nonNegLabel; (* Reach here if index non-negative *)
Asld; (* Multiply index by 2 *)
Addd (Imm (ImmWord 2)); (* Add 2 to account for index slot *)
Stx (Index(0,X)); (* Copy stack ptr to former top of stack (used as a temp) *)
Addd (Index(0,X)); (* D <- address of desired element *)
Jsr (Ext (AddrLabel "check-bounds")); (* Check for out of bounds index *)
Xgdy; (* Get here if index in bounds; put stack ref in Y *)
Ldd (Index(0,Y)); (* Load stack elt in D *)
Std (Index(0,X)); (* Put referenced element at top of stack *)
Dex; (* and push *)
Dex;]

| Put -> |
let nonNegLabel = newLabel "get-non-neg"
in gen

[Jsrb (Ext (AddrLabel "need2")); (* Ensure we at least have an index and value; leaves PFSP pointing at index *)
Ldd (Index(0,X)); (* Load index into D *)
Bge (BraLabel nonNegLabel); (* Test for negative index *)
Jmp (Ext (AddrLabel "put-neg-index")); (* Report error for negative index *)
Label nonNegLabel; (* Reach here if index non-negative *)
Asld; (* Multiply index by 2 *)
Addd (Imm (ImmWord 4)); (* Add 4 to account for index & value slots *)
Stx (Index(0,X)); (* Copy stack ptr to former top of stack (used as a temp) *)
Addd (Index(0,X)); (* D <- address of desired element *)
Jsr (Ext (AddrLabel "check-bounds")); (* Check for out of bounds index *)
Xgdy; (* Get here if index in bounds; put stack ref in Y *)
Ldd (Index(2,X)); (* Load putval in D *)
Std (Index(0,Y)); (* Store putval in referenced slot *)
Inx; (* and pop putval *)
Inx;]

Figure 10: Compiling PostFix commands, part 2.
<table>
<thead>
<tr>
<th>Add -&gt; gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Jsr (Ext (AddrLabel &quot;need2&quot;));</td>
</tr>
<tr>
<td>Ldd (Index (2, X));</td>
</tr>
<tr>
<td>Addd (Index (0, X));</td>
</tr>
<tr>
<td>Std (Index (2, X))]</td>
</tr>
<tr>
<td>Sub -&gt; gen</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>[Jsr (Ext (AddrLabel &quot;need2&quot;));</td>
</tr>
<tr>
<td>Ldd (Index (2, X));</td>
</tr>
<tr>
<td>Subd (Index (0, X));</td>
</tr>
<tr>
<td>Std (Index (2, X))]</td>
</tr>
<tr>
<td>(Div</td>
</tr>
<tr>
<td>let label = newLabel &quot;divRem&quot;</td>
</tr>
<tr>
<td>in gen</td>
</tr>
<tr>
<td>( [Jsr (Ext (AddrLabel &quot;need2&quot;));</td>
</tr>
<tr>
<td>Clr (Ext (AddrLabel &quot;sign&quot;));</td>
</tr>
<tr>
<td>Ldd (Index (2, X)); (* signed numer in D *)</td>
</tr>
<tr>
<td>Jsr (Ext (AddrLabel &quot;negate-word-by-content&quot;));</td>
</tr>
<tr>
<td>Xgdy; (* unsigned numer in Y *)</td>
</tr>
<tr>
<td>Ldd (Index (0, X)); (* unsigned numer in Y, signed denom in D *)</td>
</tr>
<tr>
<td>Cpd (Imm (ImmWord 0)); (* check for div/rem by 0 *)</td>
</tr>
<tr>
<td>Bne (BraLabel label); (* OK; proceed to calculation *)</td>
</tr>
<tr>
<td>Xgdy; (* Not OK; move unsigned numer to D *)</td>
</tr>
<tr>
<td>Jsr (Ext (AddrLabel &quot;negate-word-by-sign&quot;)); (* signed numer in D *)</td>
</tr>
<tr>
<td>Jmp (Ext (AddrLabel (if com = Div then &quot;div0-error&quot; else &quot;rem0-error&quot;))));</td>
</tr>
<tr>
<td>(* Report Error *)</td>
</tr>
<tr>
<td>Label label; (* Begin calculation *)</td>
</tr>
<tr>
<td>Jsr (Ext (AddrLabel (if com = Div then</td>
</tr>
<tr>
<td>&quot;negate-word-by-content&quot;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>&quot;negate-negative-word&quot;)));</td>
</tr>
<tr>
<td>(* Ignore sign of Denom for Rem *)</td>
</tr>
<tr>
<td>Pshx; (* Save pstk pointer! *)</td>
</tr>
<tr>
<td>Xgdx; (* Numer in X, Denom in Y *)</td>
</tr>
<tr>
<td>Xgdy; (* Numer in D, Denom in X *)</td>
</tr>
<tr>
<td>Idiv (* Quotient in X, Rem in D *)</td>
</tr>
<tr>
<td>@ (if com = Div then [Xgdx] else []) (* Put quotient or remainder in D *)</td>
</tr>
<tr>
<td>@ [Jsr (Ext (AddrLabel &quot;negate-word-by-sign&quot;)); (* signed result in D *)</td>
</tr>
<tr>
<td>Pulx; (* Restore pstk pointer! *)</td>
</tr>
<tr>
<td>Std (Index (2, X))]) (* Push result on stack *)</td>
</tr>
</tbody>
</table>

Figure 11: Compiling PostFix commands, part 3.
Figure 12: Compiling PostFix commands, part 4.
and compRel relName relFun =
  let trueLabel = newLabel (relName ^ "true")
  and joinLabel = newLabel (relName ^ "join")
in gen
  [Jsr (Ext (AddrLabel "need2"));
   Ldd (Index (2, X));
   Cpd (Index (0, X));
   relFun (BraLabel trueLabel);
   Ldd (Imm (ImmWord 0));
   Bra (BraLabel joinLabel);
   Label trueLabel;
   Ldd (Imm (ImmWord 0));
   Label joinLabel;
   Std (Index (2, X)) (* Push result on stack *)]

and compInt n =
  if (n < minInt) then
    raise (CompError ("int smaller than "
                      ^ (string_of_int(minInt))
                      ^ "; " ^ (string_of_int(n))))
  else if (n > maxInt) then
    raise (CompError ("int larger than "
                      ^ (string_of_int(maxInt))
                      ^ "; " ^ (string_of_int(n))))
  else
    gen [Ldd (Imm (ImmWord n))]

Figure 13: Compiling PostFix commands, part 5.