The Microprog. Datapath (A Practical Intro. to HW/SW Codesign, P. Schaumont)

A datapath attached to the microprogrammed controller consists of three parts:

- **Computation units** such as adders, multipliers, shifters, and so on.
- **Communications infrastructure** (bus, crossbar, point-to-point connection, etc.)
- **Storage**, typically a register file or scratchpad RAM

Each of these may contribute a few control bits to the micro-instruction word

For example,

- Multi-function computation units have selection bits that determine their specific function,
- Storage units have address bits and read/write command bits
- Communication busses have source/destination control bits

The datapath may also generate condition flags for the micro-programmed controller

The Microprogrammed Datapath

Consider the micro-programmed controller with a datapath attached

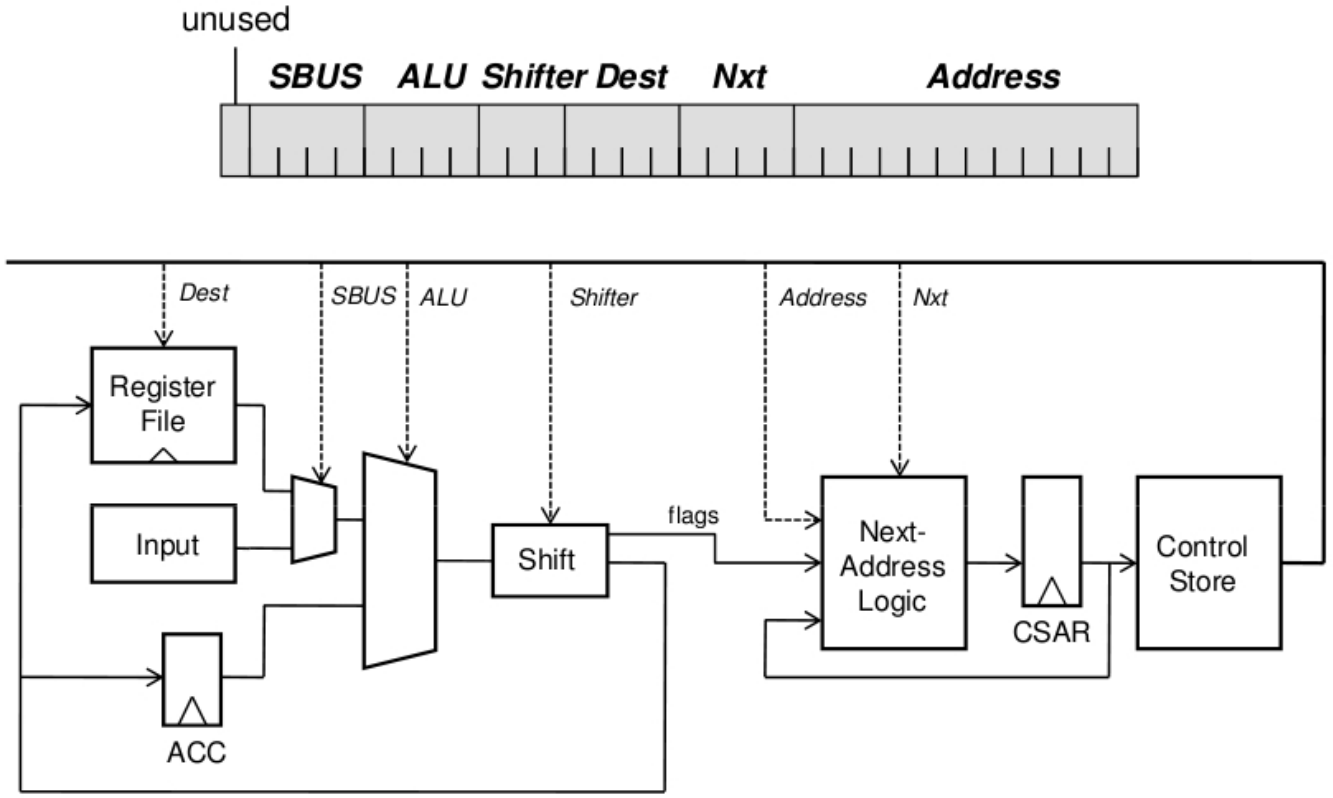

Fig. 5.7 A micro-programmed datapath

The datapath includes an ALU with shifter unit, a register file with 8 entries, an accumulator register, and an input port

The Microprogrammed Datapath

The micro-instruction word contains 6 fields

Nxt and *Address* are used by the micro-programmed **controller** while the remaining fields are used by the **datapath**

The type of encoding is **mixed** horizontal/vertical

The overall machine uses a **horizontal encoding**, i.e., each module of the machine is controlled independently

The sub-modules within the machine use a **vertical encoding**, e.g., the ALU field contains 4 bits and can execute up to 16 different commands

Other characteristics:

- The machine completes a single instruction per clock cycle
- The ALU uses operands from the accumulator, and one from the reg file or input port, and sends the result to the reg file or accumulator
- The communication used by datapath operations is controlled by 2 fields in the micro-instruction word, *SBUS* (to specify source) and *Dest* (for result)

The Microprogrammed Datapath

Other characteristics:

• The *Shifter module* also generates 2 flags, which are used by the micro-programmedcontroller to implement conditional jumps

A **zero-flag**, which is *high* when the output of the shifter is all-zero

A **carry-flag**, which is defined as the most-significant bit

Writing Micro-programs

The following table illustrates the encoding used by each module of the design

A micro-instruction defines the module function for each module of the micro-programmed machine, including a *next-address* for the *Address field*

When a field remains unused during a particular instruction, a **don't care** value can be specified

The *don't care* value are designed to prevent *unwanted state changes* in the datapath

Writing Micro-programs

Table 5.1 Micro-instruction encoding of the example machine

Field	Width	Encoding			
SBUS	4	Selects the operand that will drive the S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	Input
		0100	R4	1001	Address/Constant
ALU	4	Selects the operation performed by the ALU			
		0000	ACC	0110	ACC — S-Bus
		0001	S-Bus	0111	not S-Bus
		0010	ACC + SBus	1000	ACC + 1
		0011	ACC - SBus	1001	SBus - 1
		0100	SBus - ACC	1010	0
		0101	ACC & S-Bus	1011	1
Shifter	3	Selects the function of the programmable shifter			
		000	logical SHL(ALU)	100	arith SHL(ALU)
		001	logical SHR(ALU)	101	arith SHR(ALU)
		010	rotate left ALU	111	ALU
		011	rotate right ALU		
Dest	4	Selects the target that will store S-Bus			
		0000	R0	0101	R5
		0001	R1	0110	R6
		0010	R2	0111	R7
		0011	R3	1000	ACC
		0100	R4	1111	unconnected
Nxt	4	Selects next-value for CSAR			
		0000	CSAR + 1	1010	cf ? CSAR + 1 : Address
		0001	Address	0100	zf ? Address : CSAR + 1
		0010	cf ? Address : CSAR + 1	1100	zf ? CSAR + 1 : Address

Writing Micro-programs

For example, an instruction to copy register *R2* into the accumulator register *ACC* would be defined as follows

The instruction gets the value in register *R2* from the reg file, sends it over the *SBus*, through the *ALU* and the *shifter*, and to the *ACC* register

This functional requirement determines the values in the micro-instruction fields:

- The **SBUS** needs to transfer the value of *R2,* so (from the Table), the *SBUS* field is set to *0010*
- The **ALU** needs to pass the *SBUS* input to the output, therefore, from the Table, the *ALU* field must be set to *0001*
- The **shifter** passes the ALU output *unmodified*, thus the *Shifter* field is set to *111*
- The output of the **shifter** is used to update the **ACC**, so the *Dest* field equals *1000*
- There is no jump or control transfer for this instruction, so the *Nxt* field is set to *0000* and the *Address* field is *don't care*

The overall micro-instruction is assembled by putting all instruction fields together (as shown below) -- *0x10F80000*

Writing Micro-programs

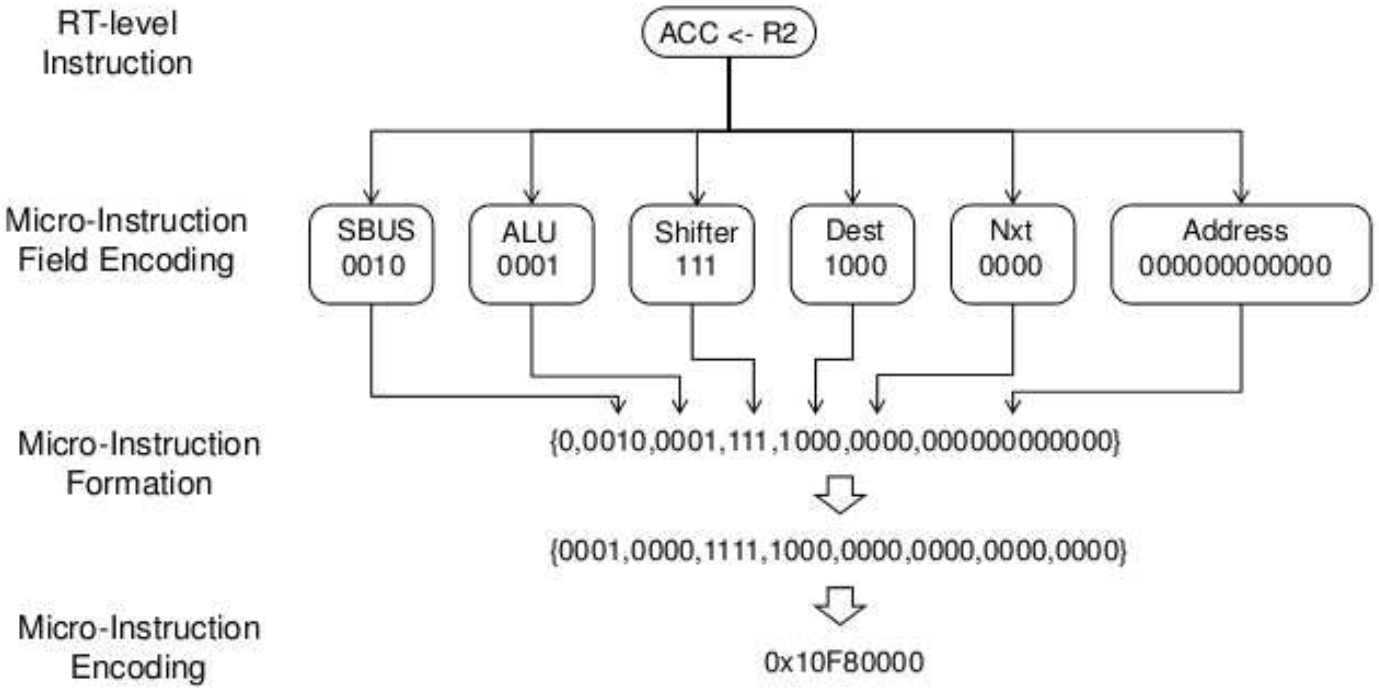

Fig. 5.8 Forming micro-instructions from Register-Transfer Instructions

Writing a micro-program thus consists of formulating the desired behavior as a *sequence of register transfers* and then encoding them in the microinstruction fields

Writing Micro-programs

Higher-level contructs, such as loops and if-then-else statements, are expressed as a **sequence** of register transfers

Although this looks like a tedious task, bear in mind that the programmer has full control over the hardware at every clock cycle

Let's write the micro-program that implements Euclid's algorithm

```
1 ;    Command Field          || Jump Field
2           IN -> R0
3           IN -> ACC
4 Lcheck: (R0 - ACC)          || JUMP_IF_Z Ldone
5         (R0 - ACC) << 1     || JUMP_IF_C LSmall
6         R0 - ACC -> R0      || JUMP Lcheck
7 Lsmall: ACC - R0 -> ACC     || JUMP Lcheck
8 Ldone:                         JUMP Ldone
```

Lines 1 and 2 read in two values from the **input port**, and store them into registers *R0* and *ACC*.

Writing Micro-programs

At the end of the program, the resulting GCD will be available in either **ACC** or **R0**

The exit condition is implemented in line 4, using a subtraction of two registers and a conditional jump based on the *zero-flag*

When the registers have different values, the program continues to subtract the largest one from the smallest one

The larger value stored in *R0* and *ACC* is determined by line 5, a conditional jump

The *bigger-than* test is implemented using a **subtraction**, a **left-shift** and a test on the resulting *carry-flag*

If the *carry-flag* is set, then the most-significant bit of the subtraction would be one, indicating a *negative* result in two's complement

This instruction is a *conditional jump-if-carry*, and is taken if *R0* is < then *ACC*

Lines 4, line 5 and line 6 implement an *if-then-else stmt* using multiple conditional and unconditional jump instructions

Implementing a Micro-programmed Machine

We implement a micro-programmed machine now in the GEZEL language

The design of a micro-programmed machine starts with defining the micro-instruction, i.e., the micro-instruction control bits, etc

```
 1 // wordlength in the datapath
 2 #define WLEN 16
 3
 4 /* encoding for data output */
 5 #define O_NIL      0     /* OT <- 0 */
 6 #define O_WR       1     /* OT <- SBUS */
 7
 8 /* encoding for SBUS multiplexer */
 9 #define SBUS_R0    0     /* SBUS <- R0 */
10 #define SBUS_R1    1     /* SBUS <- R1 */
11 #define SBUS_R2    2     /* SBUS <- R2 */
12 #define SBUS_R3    3     /* SBUS <- R3 */
13 #define SBUS_R4    4     /* SBUS <- R4 */
14 #define SBUS_R5    5     /* SBUS <- R5 */
```

Implementing a Micro-programmed Machine

```
15 #define SBUS_R6   6    /* SBUS <- R6 */
16 #define SBUS_R7   7    /* SBUS <- R7 */
17 #define SBUS_IN   8    /* SBUS <- IN */
18 #define SBUS_X SBUS_R0 /* don˜2019t care */
19
20 /* encoding for ALU */
21 #define ALU_ACC 0     /* ALU <- ACC */
22 #define ALU_PASS 1    /* ALU <- SBUS */
23 #define ALU_ADD 2      /*  ALU <- ACC + SBUS */
24 #define ALU_SUBA 3     /*  ALU <- ACC - SBUS */
25 #define ALU_SUBS 4     /*  ALU <- SBUS - ACC */
26 #define ALU_AND 5      /*  ALU <- ACC and SBUS */
27 #define ALU_OR   6     /*  ALU <- ACC or SBUS */
28 #define ALU_NOT 7      /*  ALU <- not SBUS */
29 #define ALU_INCS 8     /*  ALU <- ACC + 1 */
30 #define ALU_INCA 9     /*  ALU <- SBUS - 1 */
31 #define ALU_CLR 10     /*  ALU <- 0 */
32 #define ALU_SET 11     /*  ALU <- 1 */
```

Implementing a Micro-programmed Machine

```
33 #define ALU_X ALU_ACC  /*  don't care */
34
35 /* encoding for shifter   */
36 #define SHFT_SHL 1  /*Shifter <- shiftleft(alu) */
37 #define SHFT_SHR 2  /*Shifter <- shiftright(alu) */
38 #define SHFT_ROL 3  /*Shifter <- rotateleft(alu) */
39 #define SHFT_ROR 4  /*Shifter <- rotateright(alu) */
40 #define SHFT_SLA 5  /*Shift <- shiftleftarith(alu) */
41 #define SHFT_SRA 6  /*Shift <- shiftrightarith(alu */
42 #define SHFT_NIL 7  /*Shifter <- ALU */
43 #define SHFT_X SHFT_NIL  /*  don't care  */
44
45 /* encoding for result destination */
46 #define DST_R0  0        /* R0 <- Shifter */
47 #define DST_R1  1        /* R1 <- Shifter */
48 #define DST_R2  2        /* R2 <- Shifter */
49 #define DST_R3  3        /* R3 <- Shifter */
50 #define DST_R4  4        /* R4 <- Shifter */
```

Implementing a Micro-programmed Machine

```
51 #define DST_R5  5          /* R5 <- Shifter */
52 #define DST_R6  6          /* R6 <- Shifter */
53 #define DST_R7  7          /* R7 <- Shifter */
54 #define DST_ACC 8          /* IR <- Shifter */
55 #define DST_NIL 15         /* no connect <- shifter */
56 #define DST_X   DST_NIL /* don't care instruction */
57
58 /* encoding for command field */
59 #define NXT_NXT 0  /*CSAR<-CSAR + 1 */
60 #define NXT_JMP 1  /*CSAR<-Address */
61 #define NXT_JC  2  /*CSAR<-(carry==1)?Addr:CSAR+1*/
62 #define NXT_JNC 10  /*CSAR<-(carry==0)?Addr:CSAR+1*/
63 #define NXT_JZ   4  /*CSAR<-(zero==1)?Addr:CSAR+1*/
64 #define NXT_JNZ 12  /*CSAR<-(zero==0)?Addr:CSAR+1*/
65 #define NXT_X NXT_NXT
66
```

Implementing a Micro-programmed Machine

```
67 /* encoding for the micro-instruction word */
68 #define MI(OUT, SBUS, ALU, SHFT, DEST, NXT, ADR) \
69      (OUT  << 31)  | \
70      (SBUS << 27)  | \
71      (ALU  << 23)  | \
72      (SHFT << 20)  | \
73      (DEST << 16)  | \
74      (NXT  << 12)  | \
75      (ADR)
76
77 dp control(in  carry, zero  : ns(1);
78             out ctl_ot       : ns(1);
79             out ctl_sbus     : ns(4);
80             out ctl_alu      : ns(4);
81             out ctl_shft     : ns(3);
82             out ctl_dest     : ns(4)) {
83
```

Implementing a Micro-programmed Machine

```
84  lookup cstore : ns(32) = {
85       // 0 Lstart: IN -> R0
86  MI(O_NIL,SBUS_IN,ALU_PASS,SHFT_NIL,
       DST_R0,NXT_NXT,0),
87       // 1          IN -> ACC
88  MI(O_NIL,SBUS_IN,ALU_PASS,SHFT_NIL,
       DST_ACC,NXT_NXT,0),
89       // 2 Lcheck: (R0 - ACC)|| JUMP_IF_Z Ldone
90  MI(O_NIL,SBUS_R0,ALU_SUBS,SHFT_NIL,
       DST_NIL,NXT_JZ,6),
91       // 3          (R0 - ACC) << 1 || JUMP_IF_C LSmall
92  MI(O_NIL,SBUS_R0,ALU_SUBS,SHFT_SHL,
       DST_NIL, NXT_JC,5),
93       // 4          R0 - ACC -> R0  || JUMP Lcheck
94  MI(O_NIL,SBUS_R0,ALU_SUBS,SHFT_NIL,
       DST_R0,NXT_JMP,2),
95       // 5 Lsmall: ACC - R0 -> ACC || JUMP Lcheck
```

Implementing a Micro-programmed Machine

```
96  MI(O_NIL,SBUS_R0,ALU_SUBA,SHFT_NIL,
      DST_ACC,NXT_JMP,2),
97       // 6 Ldone: R0 -> OUT              || JUMP Lstart
98  MI(O_WR,SBUS_R0,ALU_X,SHFT_X,
      DST_X,NXT_JMP,0)
99 };
100
101  reg csar           : ns(12);
102  sig mir            : ns(32);
103  sig ctl_nxt        : ns(4);
104  sig csar_nxt       : ns(12);
105  sig ctl_address    : ns(12);
106
107  always {
108
109      mir = cstore(csar);
110      ctl_ot        = mir[31];
111      ctl_sbus      = mir[27:30];
```

Implementing a Micro-programmed Machine

```
112       ctl_alu       = mir[23:26];
113       ctl_shft      = mir[20:22];
114       ctl_dest      = mir[16:19];
115       ctl_nxt       = mir[12:15];
116       ctl_address   = mir[ 0:11];
117
118       csar_nxt = csar    + 1;
119       csar = (ctl_nxt == NXT_NXT)   ?   csar_nxt :
120          (ctl_nxt == NXT_JMP) ? ctl_address :
121          (ctl_nxt == NXT_JC) ? ((carry==1) ?
                ctl_address : csar_nxt) :
122          (ctl_nxt == NXT_JZ) ? ((zero==1) ?
                ctl_address : csar_nxt) :
123          (ctl_nxt == NXT_JNC) ? ((carry==0) ?
                ctl_address : csar_nxt) :
124          (ctl_nxt == NXT_JNZ) ? ((zero==0) ?
                ctl_address : csar_nxt) :
125           csar;
```

Implementing a Micro-programmed Machine

```
126   }
127 }
128
129 dp regfile (in  ctl_dest : ns(4);
130             in  ctl_sbus : ns(4);
131             in  data_in  : ns(WLEN);
132             out data_out : ns(WLEN)) {
133     reg r0 :  ns(WLEN);
134     reg r1 :  ns(WLEN);
135     reg r2 :  ns(WLEN);
136     reg r3 :  ns(WLEN);
137     reg r4 :  ns(WLEN);
138     reg r5 :  ns(WLEN);
139     reg r6 :  ns(WLEN);
140     reg r7 :  ns(WLEN);
```

Implementing a Micro-programmed Machine

```
141     always {
142       r0 = (ctl_dest == DST_R0) ? data_in : r0;
143       r1 = (ctl_dest == DST_R1) ? data_in : r1;
144       r2 = (ctl_dest == DST_R2) ? data_in : r2;
145       r3 = (ctl_dest == DST_R3) ? data_in : r3;
146       r4 = (ctl_dest == DST_R4) ? data_in : r4;
147       r5 = (ctl_dest == DST_R5) ? data_in : r5;
148       r6 = (ctl_dest == DST_R6) ? data_in : r6;
149       r7 = (ctl_dest == DST_R7) ? data_in : r7;
150       data_out = (ctl_sbus == SBUS_R0) ? r0 :
151                  (ctl_sbus == SBUS_R1) ? r1 :
152                  (ctl_sbus == SBUS_R2) ? r2 :
153                  (ctl_sbus == SBUS_R3) ? r3 :
154                  (ctl_sbus == SBUS_R4) ? r4 :
155                  (ctl_sbus == SBUS_R5) ? r5 :
156                  (ctl_sbus == SBUS_R6) ? r6 :
157                  (ctl_sbus == SBUS_R7) ? r7 :
158                 r0;
```

Implementing a Micro-programmed Machine

```
159   }
160 }
161
162 dp alu (in  ctl_dest  : ns(4);
163         in  ctl_alu   : ns(4);
164         in  sbus      : ns(WLEN);
165         in  shift     : ns(WLEN);
166         out q         : ns(WLEN)) {
167     reg acc : ns(WLEN);
168     always {
169        q = (ctl_alu == ALU_ACC)  ? acc :
170            (ctl_alu == ALU_PASS) ? sbus :
171            (ctl_alu == ALU_ADD)  ? acc + sbus :
172            (ctl_alu == ALU_SUBA) ? acc - sbus :
173            (ctl_alu == ALU_SUBS) ? sbus - acc :
174            (ctl_alu == ALU_AND)  ? acc & sbus :
175            (ctl_alu == ALU_OR)   ? acc | sbus :
176            (ctl_alu == ALU_NOT)  ? ~ sbus     :
```

Implementing a Micro-programmed Machine

```
177             (ctl_alu == ALU_INCS) ? sbus + 1    :
178             (ctl_alu == ALU_INCA) ? acc + 1     :
179             (ctl_alu == ALU_CLR)  ? 0           :
180             (ctl_alu == ALU_SET)  ? 1           :
181             0;
182         acc = (ctl_dest == DST_ACC) ? shift : acc;
183     }
184 }
185
186 dp shifter(in  ctl          : ns(3);
187            out zero         : ns(1);
188            out cy           : ns(1);
189            in  shft_in      : ns(WLEN);
190            out so           : ns(WLEN)) {
191   always {
192       so = (ctl == SHFT_NIL) ? shft_in :
193            (ctl == SHFT_SHL) ? (ns(WLEN))
                 (shft_in << 1) :
```

Implementing a Micro-programmed Machine

```
194              (ctl == SHFT_SHR) ? (ns(WLEN))
                    (shft_in >> 1) :
195              (ctl == SHFT_ROL) ? (ns(WLEN))
                    (shft_in # shft_in[WLEN-1]) :
196              (ctl == SHFT_ROR) ? (ns(WLEN))
                    (shft_in[0] # (shft_in >> 1)):
197              (ctl == SHFT_SLA) ? (ns(WLEN))
                    (shft_in << 1) :
198              (ctl == SHFT_SRA) ? (ns(WLEN))
                    (((tc(WLEN)) shft_in) >> 1) :
199              0;
200      zero = (shft_out == 0);
201      cy = (ctl == SHFT_NIL) ? 0 :
202              (ctl == SHFT_SHL) ? shft_in[WLEN-1] :
203              (ctl == SHFT_SHR) ? 0 :
204              (ctl == SHFT_ROL) ? shft_in[WLEN-1] :
205              (ctl == SHFT_ROR) ? shft_in[0] :
206              (ctl == SHFT_SLA) ? shft_in[WLEN-1] :
```

Implementing a Micro-programmed Machine

```
207              (ctl == SHFT_SRA) ? 0 :
208              0;
209    }
210 }
211
212 dp hmm(in din :  ns(WLEN); out din_strb  : ns(1);
213         out dout: ns(WLEN); out dout_strb : ns(1)) {
214    sig carry, zero     : ns(1);
215    sig ctl_ot          : ns(1);
216    sig ctl_sbus        : ns(4);
217    sig ctl_alu         : ns(4);
218    sig ctl_shft        : ns(3);
219    sig ctl_acc         : ns(1);
220    sig ctl_dest        : ns(4);
221
222    sig rf_out, rf_in    : ns(WLEN);
223    sig sbus             : ns(WLEN);
224    sig alu_in           : ns(WLEN);
```

Implementing a Micro-programmed Machine

```
225   sig alu_out          : ns(WLEN);
226   sig shft_in          : ns(WLEN);
227   sig shft_out         : ns(WLEN);
228
229   use control(carry,zero,
230         ctl_ot,ctl_sbus,ctl_alu,
          ctl_shft,ctl_dest);
231   use regfile(ctl_dest,ctl_sbus,rf_in,rf_out);
232   use alu (ctl_dest,ctl_alu,sbus,alu_in,alu_out);
233   use shifter(ctl_shft,zero,carry,
          shft_in,shft_out);
234
235   always {
236     sbus     = (ctl_sbus == SBUS_IN) ? din : rf_out;
237     din_strb = (ctl_sbus == SBUS_IN) ? 1 : 0;
238     dout     = sbus;
239     dout_strb = (ctl_ot == O_WR) ? 1 : 0;
240     rf_in    = shft_out;
```

Implementing a Micro-programmed Machine

```
241       alu_in    = shft_out;
242       shft_in   = alu_out;
243     }
244 }
245
246 dp hmmtest {
247     sig din          : ns(WLEN);
248     sig din_strb     : ns(1);
249     sig dout         : ns(WLEN);
250     sig dout_strb    : ns(1);
251     use hmm(din, din_strb, dout, dout_strb);
252
253     reg dcnt         : ns(5);
254     lookup stim      : ns(WLEN) =
            { 14, 32, 87, 12, 23, 99, 32, 22};
255
```

Implementing a Micro-programmed Machine

```
256     always {
257         dcnt = (din_strb) ? dcnt + 1 : dcnt;
258         din = stim(dcnt & 7);
259         $display($cycle, " IO ", din_strb, " ",
                dout_strb, " ", $dec, din, " ", dout);
260     }
261 }
262
263 system S {
264   hmmtest;
265 }
```

The listing above gives the GEZEL implementation of the micro-programmed design discussed above

The possible values for each micro-instruction field are given in Lines 5-65

The use of C macros *simplifies* the writing of micro-programs

Implementing a Micro-programmed Machine

The definition of a single micro-instruction is done using a C macro as well, shown in Lines 68-75

Lines 78-128 show the micro-programmed **controller**, which includes a control store with a micro-program and the next-address CSAR logic

The **control store** is a *lookup table* with a sequence of micro-instructions (lines 85-100)

On line 110, a micro- instruction is fetched from the control store, and broken down into individual fields

These define the output of the microprogrammed controller (lines 110-117)

The *next-address* logic uses the next-address control field to determine a new value for CSAR each clock cycle (lines 120-126)

The micro-programmed machine includes several **data-paths**, including a *register file* (lines 130-161), an *ALU* (lines 163-185), a *shifter* (lines 187 - 211)

Implementing a Micro-programmed Machine

Each of the data-paths is crafted along a similar principle

Based on the control field input, the data-input is transformed into a corresponding data-output

The decoding process of control fields is defined as a sequence of ternary selection-operators

The top-level cell for the micro-programmed machine is shown in lines 213-245

The top-level includes the **controller**, a **register file**, an **ALU** and a **shifter**

The top-level module also defines a **data-input port** and a **data-output port**, and each has a strobe control signal that indicates a data-transfer

The strobe signals are generated by the top-level module based on the decoding of micro-instruction fields

The input strobe is generated when the *SBUS* control field indicates that the *SBUS* will be reading an external input

Implementing a Micro-programmed Machine

The output strobe is generated by a separate, dedicated micro-instruction bit

A simple testbench for the top-level cell is shown on lines 247-266

The testbench feeds in a sequence of data to the micro-programmed machine, and prints out each number appearing at the data output port

The micro-program for this machine evaluates the GCD of each tuple in the list of numbers shown on line 255

This design can be simulated with the fdlsim GEZEL simulator

The C macro's require that the program first be pre-processed -- following this, the code is simulated for 100 cycles:

```
>cpp -P hmm2.fdl | fdlsim 100
```

The first few lines of the output:

```
0 IO    1    0    14   14
1 IO    1    0    32   32
2 IO    0    0    87   14
```

Implementing a Micro-programmed Machine

```
3  IO    0    0    87   14
4  IO    0    0    87   14
...
```

The micro-programmed machine reads the numbers 14 and 32 in cycle 0 and 1, and starts the GCD caculculation

To find the corresponding GCD, we look for a '1' in the fourth column (output strobe), which happens around cycle 21 to yield GCD(32,14) = 2

```
18   IO    0    0   87   2
19   IO    0    0   87   2
20   IO    0    0   87   2
21   IO    0    1   87   2
22   IO    1    0   87   87
23   IO    1    0   12   12
24   IO    0    0   23   87
```

Implementing a Micro-programmed Machine

A quick command to filter out the valid outputs during simulation is the following

```
cpp -P hmm2.fdl | fdlsim 200 | awk '{if ($4 == "1")
print $0}'
21 IO 0 1 87 2
55 IO 0 1 23 3
92 IO 0 1 32 1
117 IO 0 1 14 2
139 IO 0 1 87 2
173 IO 0 1 23 3
```

The above design illustrates how the FSMD model can be applied to create a more complex micro-programmed machine

In the following, we show how this can be used to create programming concepts at even higher levels of abstraction, using **micro-program interpreters**