Micro-program Interpreters (A Practical Intro. to HW/SW Codesign, P. Schaumont)

A micro-program is a highly-optimized sequence of commands (optimized for parallelism) for a datapath

Writing **efficient** micro-programs requires an in-depth understanding of the machine architecture

A common usage of micro-programs is to serve as **interpreters** for other programs, and **not** to encode complete applications

An interpreter is a machine that decodes and executes instruction sequences of an abstract high-level machine -- a **macro-machine**

The instructions from the macro-machine will be implemented as micro-programs

A micro-program interpreter is designed as an infinite loop

It reads a macro-instruction byte and decodes it into opcode and operand fields

It then takes specific actions depending on the values of the opcode

Micro-program Interpreters

A micro-program interpreter

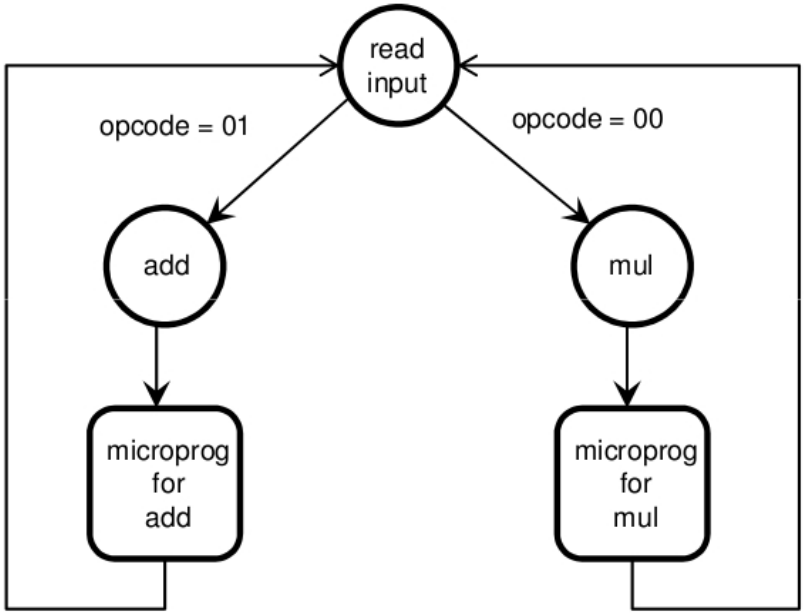

Fig. 5.9 A micro-program interpreter implements a more abstract language

Consider the following simple machine as a programmers' model of the macro-machine

It has four registers *RA* through *RD*, and two instructions for adding and multiplying those registers

Micro-program Interpreters

The macro-machine has the same wordlength as the micro-programmed machine but has fewer register than the micro-programmed machine

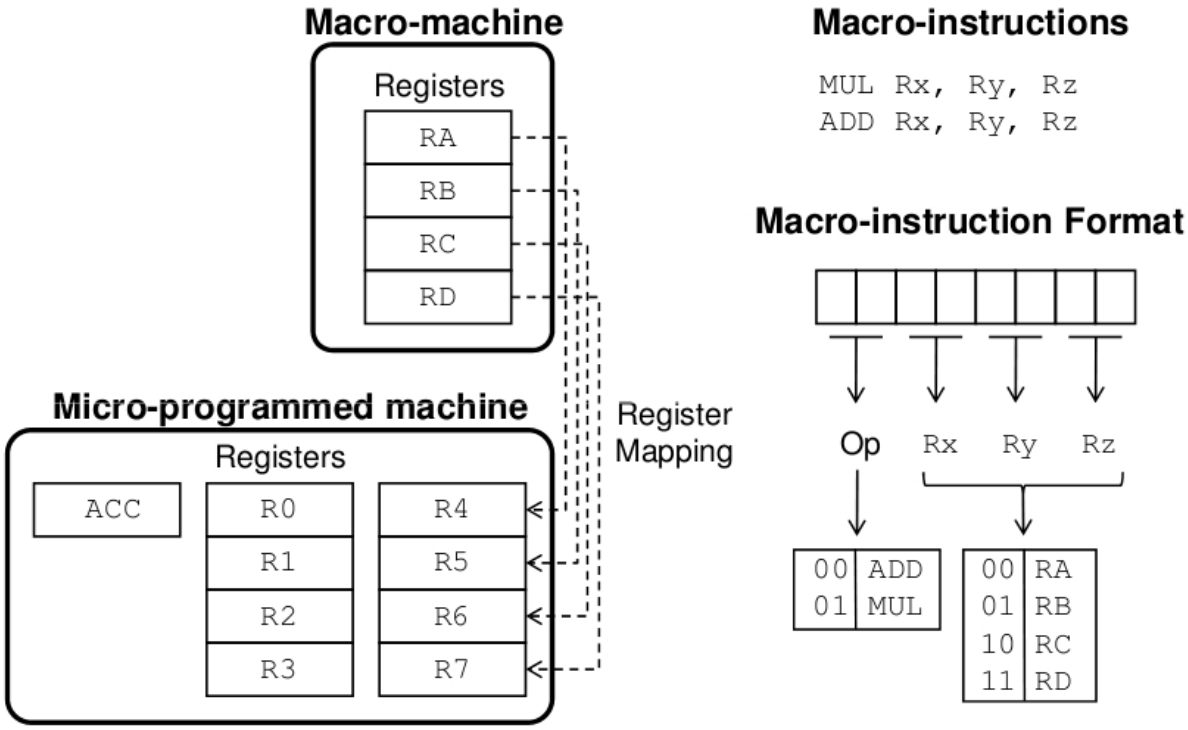

Fig. 5.10 Programmer's model for the macro-machine example

To implement the macro-machine, we map the macro-register set directly onto the micro-register set (as shown above)

Micro-program Interpreters

This leaves register *R0* to *R3*, and the *accumulator*, available to implement macro-instructions

The macro-machine has two instructions: *ADD* and *MUL*, which take two source operands (in the macro-machine registers) and generates one

The micro-machine needs a **decoder** for macro-instructions (which are 1 byte wide)

The format is two bits for the macro-opcode, and two bits for each of the macro-instruction operands

Consider the following implementation the ADD and MUL instructions:

```
1 //-----------------------------------------
2 // Macro-machine for the instructions
3 //
4 //    ADD Rx, Ry, Rz
5 //    MUL Rx, Ry, Rz
6 //
```

Micro-program Interpreters

```
 7 //    Macro-instruction encoding:
 8 //    +----+----+----+----+
 9 //    | ii + Rx + Ry + Rz +
10 //    +----+----+----+----+
11 //
12 //    where ii = 00 for ADD
13 //               01 for MUL
14 //    where Rx, Ry and Rz are encoded as follows:
15 //               00 for RA (mapped to R4)
16 //               01 for RB (mapped to R5)
17 //               10 for RC (mapped to R6)
18 //               11 for RD (mapped to R7)
19 //
20 // Interpreter loop reads instructions from input
21 macro:    IN -> ACC
22          (ACC & 0xC0) >> 1 -> R0 // shift 6 right
23          R0 >> 1 -> R0           // most bits off
24          R0 >> 1 -> R0
```

Micro-program Interpreters

```
25              R0 >> 1 -> R0
26              R0 >> 1 -> R0
27              R0 >> 1 -> R0       || JUMP_IF_NZ mul
28              (no_op)             || JUMP add
29 macro_done: (no_op)              || JUMP macro
30
31 //-----------------------------------------------
32 //
33 // Rx = Ry + Rz
34 //
35 add:          (no_op)             || CALL getarg
36              ACC -> R0
37              R2 -> ACC
38              (R1 + ACC) -> R1
39              R0 -> ACC           || CALL putarg
40              (no_op)             || JUMP macro_done
41
```

Micro-program Interpreters

```
42 //-----------------------------------------------
43 //
44 // Rx = Ry * Rz
45 //
46 mul:          (no_op)           || CALL getarg
47               ACC -> R0
48               0   -> ACC
49               8   -> R3
50 loopmul:     (R1 << 1) -> R1 || JUMP_IF_NC nopartial
51               (ACC << 1) -> ACC
52               (R2 + ACC) -> ACC
53 nopartial: (R3 - 1) -> R3    || JUMP_IF_NZ loopmul
54               ACC -> R1
55               R0 -> ACC         || CALL putarg
56               (no_op)           || JUMP macro_done
57
58 //-----------------------------------------------
59 //
```

Micro-program Interpreters

```
60 // GETARG
61 //
62 getarg:     (ACC & 0x03) -> R0 || JUMP_IF_Z Rz_is_R4
63              (R0 - 0x1)         || JUMP_IF_Z Rz_is_R5
64              (R0 - 0x2)         || JUMP_IF_Z Rz_is_R6
65 Rz_is_R7: R7 -> R1               || JUMP get_Ry
66 Rz_is_R6: R6 -> R1               || JUMP get_Ry
67 Rz_is_R5: R5 -> R1               || JUMP get_Ry
68 Rz_is_R4: R4 -> R1               || JUMP get_Ry
69 get_Ry:   (ACC & 0x0C) >> 1 -> R0
70            R0 >> 1 -> R0         || JUMP_IF_Z Ry_is_R4
71           (R0 - 0x1)            || JUMP_IF_Z Ry_is_R5
72           (R0 - 0x2)            || JUMP_IF_Z Ry_is_R6
73 Ry_is_R7:  R7 -> R2              || RETURN
74 Ry_is_R6:  R6 -> R2              || RETURN
75 Ry_is_R5:  R5 -> R2              || RETURN
76 Ry_is_R4:  R4 -> R2              || RETURN
77
```

Micro-program Interpreters

```
78 //-----------------------------------------------
79 //
80 // PUTARG
81 //
82 putarg:    (ACC & 0x30) >> 1 -> R0
83              R0 >> 1 -> R0
84              R0 >> 1 -> R0
85              R0 >> 1 -> R0       || JUMP_IF_Z Rx_is_R4
86             (R0 - 0x1)           || JUMP_IF_Z Rx_is_R5
87             (R0 - 0x2)           || JUMP_IF_Z Rx_is_R6
88 Rx_is_R7: R1 -> R7                || RETURN
89 Rx_is_R6: R1 -> R6                || RETURN
90 Rx_is_R5: R1 -> R5                || RETURN
91 Rx_is_R4: R1 -> R4                || RETURN
```

The micro-interpreter loop, on line 21-29, reads one macro-instruction from the input, *IN*, and stores it in the *ACC* register

It determines the macro-instruction opcode with a couple of shift instructions

Micro-program Interpreters

The opcode field determines whether the micro-program jumps to *ADD* or *MUL* routine (We assume that single-level calls to subroutines are supported)

Macro-instructions can use one of four possible operand registers -- therefore, an additional **register-move** operation, *putarg* and *getarg*, is needed

getarg subroutine copies data from the macro-machine source registers (RA through RD) to the micro-machine source working registers (R1 and R2)

putarg subroutine moves data from the micro-machine destination **working register** R1 back to the destination macro-machine register (RA through RD).

Note that the implementation of *MUL* preserves only the lower order byte of the product (need 16-bits for 2 8-bit operands)

A micro-programmed interpreter can create the illusion of a machine that has more powerful instructions than the original micro-programmed architecture

Bear in mind the performance impact introduced by the many-to-one mapping

Micro-program Interpreters

The concept of micro-program interpreters has been used extensively to design processors with **configurable instruction sets**

And it was originally used to enhance the flexibility of expensive hardware

Today, the technique of micro-program interpreter design is still very useful for creating an additional level of abstraction on top of a micro-programmed architecture

Micro-program Pipelining

Pipeline registers can be used to break up the micro-program controller logic

However, adding pipeline registers has a large impact on the design of micro-programs

First consider that the CSAR register (next slide) is part of possibly three combinational logic loops

- First loop runs through the next-address logic
- Second loop runs through the control store and the next-address logic
- Third loop runs through the control store, the data path, and the next-address logic

Micro-program Pipelining

These combinational paths may limit the maximum clock frequency of the micro-programmed machine

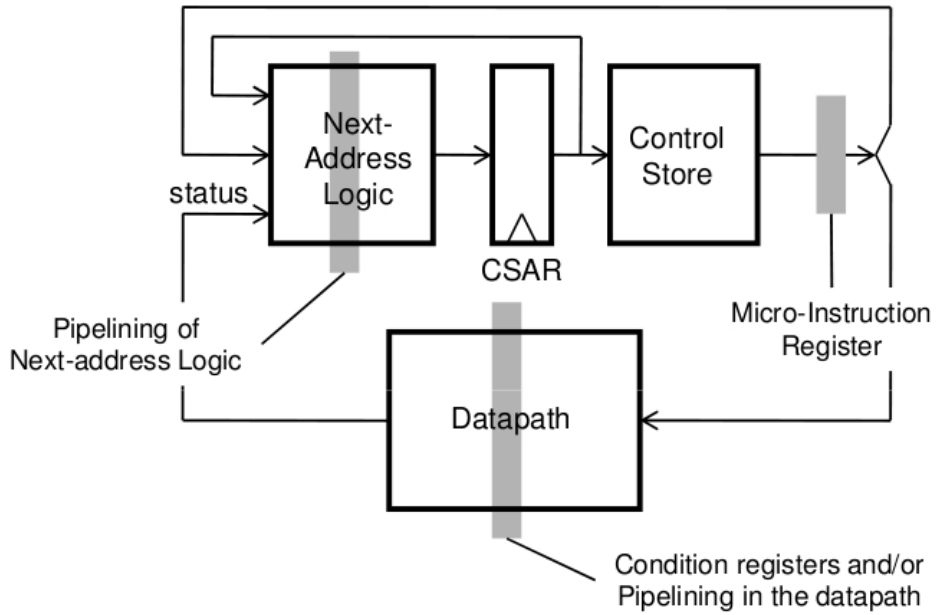

Fig. 5.11 Typical placement of pipeline registers in a micro-program interpreter

There are three common places where **pipeline registers** may be inserted, as shown above with shaded boxes

- At the output of the control store: as a **micro-instruction register**

  Inserting a register there allows overlap of the datapath evaluation, the next address evaluation, and the micro-instruction fetch

Micro-program Pipelining

- In the datapath

  Also, additional condition-code registers can be inserted on datapath outputs

- For the next-address logic

  For high-speed operation when the target CSAR address cannot be evaluated within a single clock cycle

Micro-instruction Register

Note that each of these registers cuts through a different update-loop of the CSAR register

Therefore, each of them will have a different effect on the micro-program

Consider the effect of adding the micro-instruction register

With this register in place, the micro-instruction fetch is **offset by one cycle** from the evaluation of that micro-instruction

For example, when the CSAR is fetching instruction *i*, the datapath and next-address logic will be executing instruction *i - 1*

Micro-program Pipelining

Consider this *offset* under the condition that the instruction stream contains a jump instruction

Table 5.2 Effect of the micro-instruction register on jump instructions

Cycle	CSAR	Micro-instruction Register
N	4	
N+1	5	CSTORE(4) = JUMP 10
N+2	10	CSTORE(5) *need to cancel*
N+3	11	CSTORE(10) *execute*

The micro-programmer entered a *JUMP 10* instruction in CSTORE location 4, and which is fetched in clock cycle *N*

In clock cycle *N+1*, the micro-instruction will appear at the output of the micro-instruction register and its execution will complete in cycle *N+2*

For a JUMP, this means that the CSAR should NOT point to the next instruction in cycle *N+2*, but the instruction at *N+2* has already been fetched

This instruction needs to be **canceled**

Micro-program Pipelining

There are two ways to deal with this problem:

- Programmer takes into account that a JUMP will be executed with one cycle of delay (so-called '*delayed branch*')
- Include support in the micro-programmed machine to cancel the execution of an instruction in case of a jump

Datapath Condition-Code Register

Assume that we have a condition-code register in the datapath, in addition to a micro-instruction register

The fact that its a register means that the actual condition code **will not** be available in the current clock cycle (when the expression is evaluated)

Therefore, conditional-jump instructions can only operate on datapath conditions from the previous clock cycle

Datapath Condition-Code Register

Table 5.3 Effect of the micro-instruction register and condition-code register on conditional jump instructions

Cycle	CSAR	Micro-instruction Register
N	3	
N+1	4	CSTORE(3) = TEST R0 sets Z-flag
N+2	5	CSTORE(4) = JZ 10
N+3	10	CSTORE(5) *need to cancel*
N+4	11	CSTORE(10) *execute*

Here, the branch instruction in CSTORE(4) is a conditional jump

When the condition is true, the jump will be executed with **one clock cycle delay**

The *JZ* instruction implements the jump in cycle *N+2*, which tests the condition code generated in cycle *N+1* and which becomes available in *N+2*

Here, the micro-programmer just needs to be aware that condition flag need to be generated one clock cycle before they are used in conditional jumps

Note instruction at *N+3* needs to be cancelled if jump is taken

Pipelined Next-address Logic

Assume that there is a third level of pipelining available inside of the next-address update loop

For simplicity, we will assume there are two CSAR registers back-to-back in the next-address loop

The output of the next-address-logic is fed into the CSAR pipeline register
And the output of CSAR pipeline register is connected to CSAR

Assuming all registers are initially zero, the two CSAR registers in the next-address loop result in **two (independent) address sequences**

Table 5.4 Effect of additional pipeline registers in the CSAR update loop

Cycle	CSAR_pipe	CSAR	Micro-instruction Register
0	0	0	CSTORE(0)
1	1	0	CSTORE(0) twice ?
2	1	1	CSTORE(1)
3	2	1	CSTORE(1) twice ?

Pipelined Next-address Logic

We see that each instruction of the micro-program is executed twice!

What is needed is a careful initialization of CSAR pipe and CSAR such that they start out at different values (e.g. 1 and 0)

Unfortunately, this needs to be done for each jump instruction too

This complicates the design and the programming of pipelined next-address logic

These examples show that a micro-programmer **must be aware** of the implementation details of the micro-architecture

In particular, he/she MUST be aware of all the delay effects caused by registers

This significantly increases the complexity of the development of micro-programs

Picoblaze: A Contemporary Microprogram Controller

When complex systems are created in hardware, the design of an adequate control architecture is often a key problem

In this section, we illustrate a possible solution based on the use of a micro-controller

Most FPGA companies now offer small programmable, synthesizable controllers

This includes for example **Picoblaze** (Xilinx), **Mico8** (Lattice Semiconductor) or **Avalon** (Altera)

These controllers have only minimal computational capabilities, such as an ALU with basic logical and arithmetic operations

However, they do implement an **instruction-fetch engine**, and as such as they are well suited as controllers for larger circuits

They also come with a **pseudo-assembly instruction-set**, that allows for easy design of control programs

For these reasons, these controllers are well suited as replacement for FSMs

Picoblaze: A Contemporary Microprogram Controller

Here, we consider another use of these controllers, namely using them as **micro-program controllers**

Picoblaze

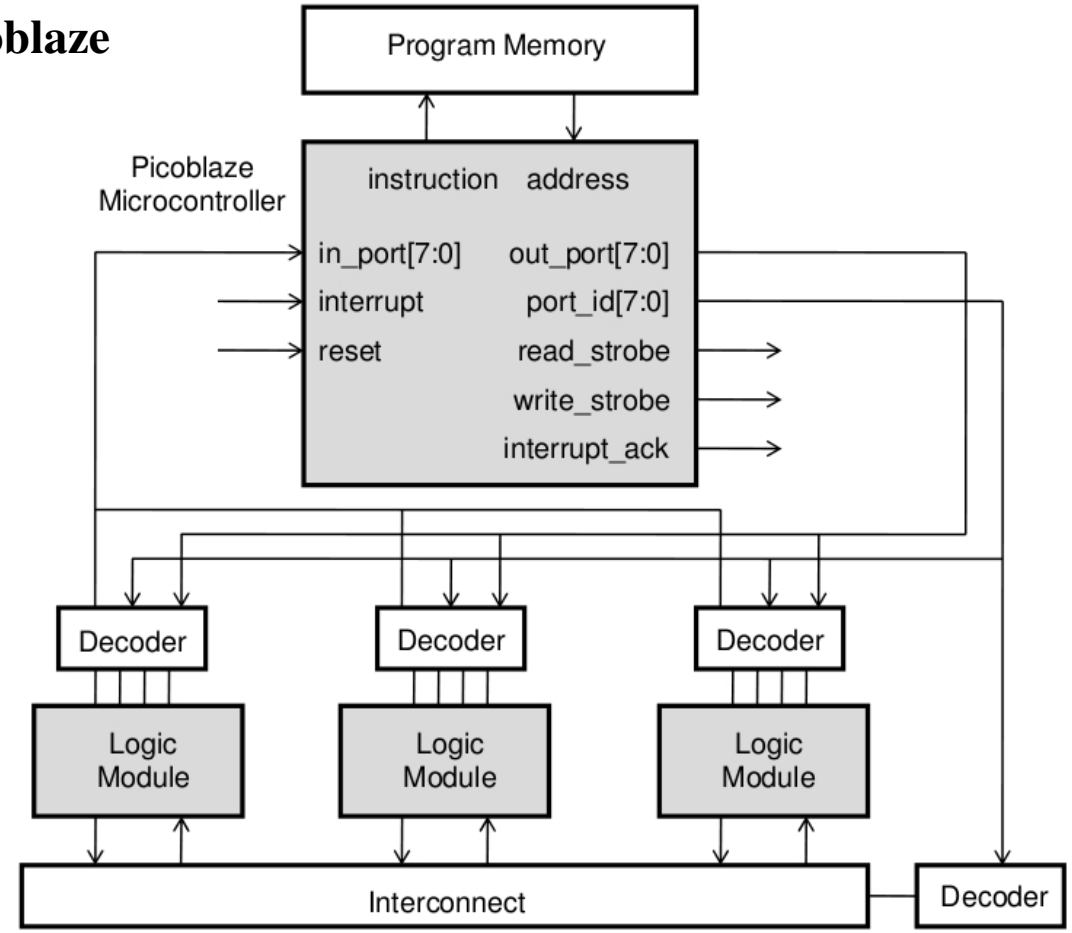

Fig. 5.12 Using a Picoblaze controller as a microprogram sequencer

Picoblaze: A Contemporary Microprogram Controller

The Picoblaze controller is an 8-bit architecture with an internal program memory

The controller has several **additional ports** that are helpful for using this module as a system controller

- An 8-bit input port `in_port` can read in 8-bit values
- An 8-bit output port `out_port` can produce 8-bit values
- A port identifier `port_id` carries a port address

  This allows the picoblaze controller to distinguish 256 multiplexed input and output ports
- A `read_strobe` and `write_strobe` synchronizes input/output operations on the I/O ports
- Additional control lines will initialize the controller (`reset`) or will handle interrupts (`interrupt`, `interrupt_ack`)

The Picoblaze controller has several instructions to communicate through I/O ports

```
OUTPUT  sX,  sY; // write contents of
                 // reg sX to port ID reg sY
```

Picoblaze: A Contemporary Microprogram Controller

```
OUTPUT  sX,  kk; // write contents of reg
                 // sX  to port ID const kk
INPUT   sX,  sY;  // read contents of port ID
                  // reg sY into reg sX
INPUT   sX,  kk;  // read contents of port ID
                  // const kk into reg sX
```

In the figure above, I/O ports are used to **control several datapath sub-modules**

Combining the port address and the port data, we can communicate up to **16 bits** of data per Picoblaze instruction to the datapath sub-modules

Thus, we need to use a *vertically-encoded micro-instruction* to accommodate a large number of logic modules

This is shown by the decoders on top of each logic module

Also, there can be up to **8 bits** of *status information* feed back to the Picoblaze controller