The picoProcessor VHDL Models

Introduction

These notes describe the VHDL models for the picoProcessor. The instruction set architecture is described in the document The picoProcessor ISA. The VHDL model suite includes:

- A behavioral model
- A model of an unpipelined implementation that performs each instruction in a single clock cycle
- A model of an unpipelined implementation that performs each instruction in several clock cycles.

The Definitions Package

The `pP_defs` package defines types and constants used in the model suite. The package declaration is in the file *pP_defs.vhd*, and the package body is in *pP_defs-body.vhd*.

Definitions Package Declaration

The type `byte` is the basic data type for pP instructions, and `instruction` is the 19-bit encoded instruction type. `Byte_array` and `instruction_array` are used for data and instruction memories, respectively. `Std_ulogic_byte` and `std_logic_byte` are used for bidirectional data buses. Such buses may have multiple sources and so need to be driven with tristate drivers. Hence, standard-logic vectors are used for them.

The group of subtypes from `instruction_addr` to `shift_count` represent addresses and values that are used within the models and encoded in instructions. The function-code subtypes (`..._fn_code`) represent fields encoded in instructions and used to select operations from different instruction groups. The constants `alu_fn_add` to `branch_fn_bnc` represent the binary encoding for the different operation function codes.

The group of subtypes `op2` to `op6` represent the opcode fields of instructions, and the constants that follow represent the corresponding binary encodings.

The string subtype `disassembled_instruction` and the procedure `disassemble` are used for debugging the VHDL models. `Disassemble` takes an instruction and yields a textual representation, showing the instruction name and operands in disassembled form.

The `IMem_array` subtype is used for the instruction memory of the pP, and the `load_program` function is used to read the initial contents of the instruction memory from a file.

Definitions Package Body

The package body contains implementations of the `disassemble` and `load_program` subprograms.

`Disassemble` makes use of a string, `result`, to form the disassembled instruction text. It initializes the string to all spaces then fills in pieces, depending on the kind of instruction. It uses aliases for slices of the instruction word to identify the opcodes, function codes, register addresses and other fields.

The steps for a register-register ALU instruction illustrate how disassembly proceeds. First, a name table is indexed with the function code to select the instruction mnemonic, which is copied into the beginning of the `result` string. Next, since register-register instructions refer to three register numbers, the ‘R’ characters for the register names are filled in. Then, for each register operand, the `disassemble_reg` procedure is called to fill in the register number. The second parameter indicates the position in the `result` string for the register number. The `disassemble_reg` procedure simply converts the binary-encoded register number to integer form and uses the `'image` attribute to obtain a textual representation.

Disassembling the other kinds of instructions is similar. In each case, the function code is used to index the appropriate name table, and subprograms are called to disassemble the instructions fields. In the case of disassembling the effective address for a memory/IO instruction, the base register number is checked to see if direct (base register is r0) or displacement (base register is not r0) addressing is used. In the former case, the displacement is treated as an unsigned 8-bit address and the register number is not included. Otherwise, the displacement is treated as a signed 8-bit number and the base register number is disassembled.

The `load_program` function reads lines from a file to initialize an instruction store. Each line contains an address written in numeric form, followed by white space and then a binary encoded instruction. Any text on the line after the instruction is ignored and can be treated as a comment.

The function initializes the instruction store in `result` to all zeros. Provided the file-name parameter string is non-empty, the function uses the string to open the file. It then reads lines until the end of the file is reached. For each line, the function extracts the address and the bit-vector form of the instruction. It type-converts the bit-vector form to the `instruction` type and checks that the extracted address is within the valid address range. If so, the function stores the instruction at the address in the result. When the end of the file is reached, the function closes the file and returns the result.

The pP Entity

The file *pP.vhd* contains the entity declaration for the pP. The generic list includes a string for specifying the name of the file from which to initialize the instruction memory and a boolean flag for controlling whether debugging messages are issued.

The port list contains the external interface of the pP. The `clk` port is the master clock governing timing of the pP. All other ports are sampled or set synchronously with the clock. The `reset` port re-initializes the pP to its reset state. When `reset` is negated, the pP commences instruction execution.

The ports `port_addr` to `port_ready` represent the interface to I/O port registers. `Port_addr` identifies a particular port to access, and `port_data` carries the data written or read. `Port_write` is activated to write to a port, and `port_read` is activated to read from a port. The port controller must activate `port_ready` when it has accepted write data or provided read data. The timing of read and write operations varies between implementations of the pP.

The `int_req` port is used to request an interrupt of the pP. When this port is active and the pP interrupts are enabled, the pP will save state and transfer to the interrupt service code. It asserts `int_ack` for one cycle to indicate start of interrupt service. The port controller must negate `int_req` before the service code returns and re-enables interrupts; otherwise a second spurious interrupt will be received. Usually, a port controller would negate the interrupt request in response to `int_ack` or to the pP reading or writing a port register.

The pP Behavioral Architecture

The file *pP-behav.vhd* contains a behavioral architecture body, named `behav`, for the pP. The architecture contains a single process, `interpreter`, that contains variables representing the machine state and sequential code to execute instructions. The code executes one instruction per clock cycle, except that port input and output operations may extend over multiple cycles if the port controller is not immediately ready.

Within the `interpreter` process, the constant `IMem` represents the instruction memory, initialized using the `load_program` function from the `pP_defs` package. The variable `DMem` represents the data memory. Various other variables represent other parts of the machine state:

- `PC` is the program counter.
- `IR` is the instruction register, containing an instruction fetched from `IMem` to be executed.
- `stack` is the return-address stack for subroutine calls.
- `SP` is the stack pointer for the return-address stack.
- `GPR` is the general purpose register file, containing registers r0 to r7.
- `cc_c` and `cc_z` are the condition code bits.
- `int_en` is the interrupt enable bit.
- `int_PC`, `int_z` and `int_c` represent the interrupt register, in which the PC and condition code bits are saved during interrupt service.

The aliases (`IR_...`) are used to refer to fields of the instruction register. The variables `disassembled_instr` and `debug_line` are used to form debug lines when tracing operation of the pP.

The body of the interpreter process starts by calling the `perform_reset` procedure to reset the pP. That procedure sets all of the output ports to their inactive state and zeros the machine state. The interpreter then waits until the `reset` port is negated, then enters the main loop for fetching and executing instructions.

Each iteration of the main loop involves waiting for a clock edge. If `reset` is active, the loop is exited and the process repeats from the beginning. Otherwise, if an interrupt request is pending (interrupts enabled and `int_req` active), the process acknowledges the request by calling the `perform_interrupt` procedure then continuing with the next main-loop iteration.

The `perform_interrupt` procedure first converts the current PC value to integer form for use in a debugging message. It then saves the current PC and condition code bits in the interrupt-register variables, activates the `int_ack` port and sets the PC to 1 (the address of the interrupt service code). Finally, if the `debug` generic is true, the procedure forms and writes a debug message using the converted PC value.

The interpreter main loop, if there is no pending interrupt, next fetches an instruction using the `fetch_instruction` procedure. That procedure negates the `int_ack` port, converts the current PC value to integer form, uses the converted value to read an instruction from the `IMem` array, then increments the PC variable. If the `debug` generic is true, the procedure then disassembles the fetched instruction and forms and writes a debug message using the original PC value and the disassembled instruction string.

Having fetched an instruction, the interpreter process then examines the opcode fields to determine how to execute the instruction. In each case, the process calls a subordinate procedure to perform the required actions. In the case of ALU and shift instructions, the subordinate procedure is only called if the destination register is not r0, since r0 must retain its reset value of 0.

The `perform_alu_op` procedure is used for both register-register and register-immediate instructions. Depending on which class of instruction is being executed, different operands are passed to the procedure. The procedure uses the function code to select which operation to perform. In each case, the operation is performed on 9-bit zero-extended versions of the operands so that the carry-out bit can be determined. The procedure's result is the least-significant eight bits of the 9-bit result. The zero flag is determined by comparing the 8-bit result with zero, and the carry flag is taken from the left-most bit of the 9-bit result. For addition and subtraction with carry, the numeric value of the carry bit is determined by using the position number of the bit (0 for '0' and 1 for '1').

The `perform_shift` procedure uses the shift function code to select which operation to perform. For left and right shifts, the 8-bit operand is zero-extended on the side where bits are shifted out. The extended bit-position becomes the carry out, and the other eight bits form the 8-bit result. For rotate operations, the 8-bit operand value is rotated without extension. The carry out for left rotates is the rightmost bit of the result, since that is the bit that was rotated out of the left end of the operand value. Similarly, for right rotates, the carry out bit is the leftmost bit of the result. For all operations, the zero flag is determined by comparing the 8-bit result with zero.

The `perform_mem` procedure starts by calculating the effective address using the `perform_alu_op` procedure. The parameters passed to the procedure are the function code for performing an addition, the base-register value, the displacement and a carry-in of zero. The byte result is used as the memory address, and the condition-code flags are ignored. The `perform_mem` procedure then uses the memory function code to select the memory or I/O operation to perform. For a memory load, provided the destination register is not r0, the procedure copies a byte from the `DMem` variable to the destination register. For a memory store, the procedure copies a byte in the reverse direction. For an input instruction, the procedure assigns the effective address to `port_addr` and activates the `port_read` control signal. It then loops, waiting for successive clock edges. If the `reset` input is activated, the procedure returns immediately, letting the interpreter main loop deal with the reset condition. Otherwise, if the `port_ready` input is active, the procedure exits the inner loop and copies the byte from `port_data` to the destination register (provided the destination register is not r0). The procedure then resets `port_addr` to zero and negates `port_read`. The actions for an output instruction are similar to those for an input instruction. The differences are that the data from the source register is converted to standard-logic form and assigned to `port_data` and the `port_write` signal is activated instead of `port_read`.

The `perform_branch` procedure uses the branch function code to determine which condition code to test and for which value. Depending on the outcome of the test, the procedure sets the `branch_taken` variable. If the branch is taken, the procedure updates the PC by adding the signed displacement to it.

The `perform_jump` instruction uses the op5 field of the instruction register to select the form of jump. For a jump instruction, the procedure simply copies the target address from the IR to the PC. For a jump-to-subroutine instruction, the procedure first saves the PC in the return-address stack and increments the stack pointer. It then copies the target address to the PC.

The remaining instructions are performed by the `perform_misc` procedure, which uses the op6 field of the IR to selection the action. For a return from subroutine instruction, the procedure decrements the stack pointer and restores the saved address to the PC. For a return from interrupt instruction, the procedure restores the saved address and condition code bits and re-enables interrupts. For the enable and disable interrupt instructions, the procedure sets or clears the `int_en` bit appropriately.

An Unpipelined Single-Cycle Organization

Figure 1 shows an unpipelined pP organization that executes each instruction in one clock cycle. Values stored on one clock edge flow through the data path, and the machine state is updated on the next clock edge. The clock period must be long enough for the slowest path through the design.

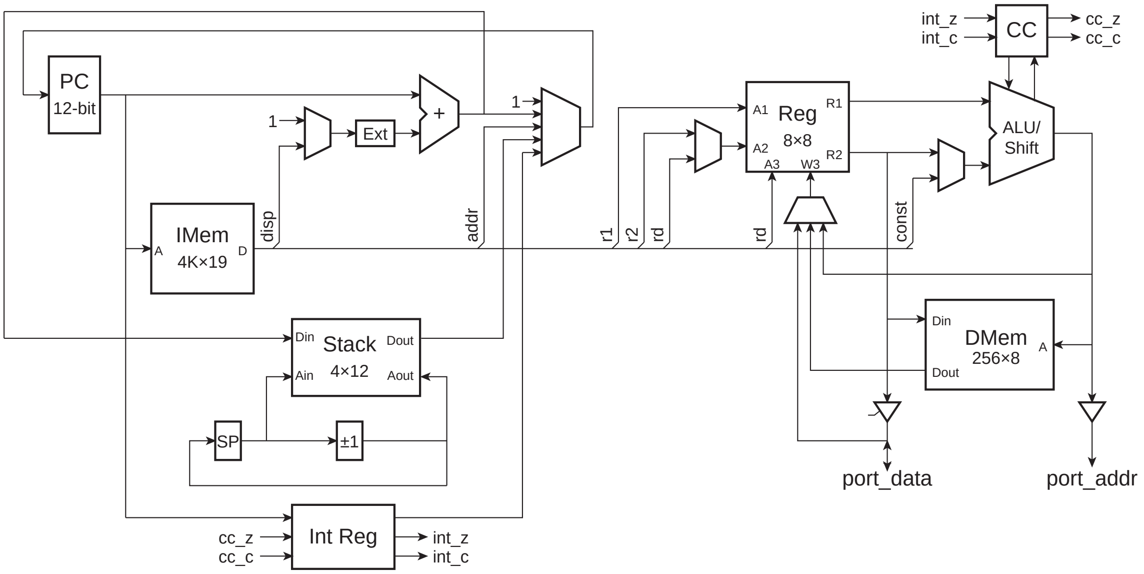

FIGURE 1 Unpipelined, single-cycle organization.

Execution within a cycle starts with checking whether an interrupt request is pending. If one is, the current PC and condition code bits are saved in the interrupt register and the next PC value is selected to be the address 1. No other machine state is updated.

If no interrupt is requested, the PC value is used to index the instruction memory to fetch the instruction to be executed. Since all operations for the instruction take place within a cycle, the instruction memory must be an asynchronous ROM. The next PC value depends on the instruction opcode and, in the case of branch instructions, whether the branch is taken or not. For `JSB` instructions, the next PC value is saved into the return-address stack and the stack pointer is incremented. For `RET` instructions, the top value in the stack is used as the next PC and the stack pointer is decremented.

The general-purpose register (GPR) file is an a multiport register file with two asynchronous read ports and a synchronous write port. The register address field for one read port is the r1 field of the instruction. The address for the other read port is either the rd field (for `STM` and `OUT` instructions) or the r2 field (for other instructions). The write port is used for ALU, shift, load and input instructions, provided the destination address is not r0. The rd field from the instruction is used as the write-port address, and the data to be written comes from the appropriate source, depending on the instruction.

The ALU calculates the result value for ALU and shift instructions and the effective address for memory instructions. For ALU and shift instructions, the condition code bits are updated according to the result. The multiplexer on the ALU input selects between a register operand for register-register instructions or the constant value from the instruction for immediate and memory instructions.

The data memory is asynchronously read for load instructions and synchronously written for store instructions. The address comes from the ALU result.

The external port interface is used for input and output instructions. Since this implementation of the pP executes instructions within a single cycle, it assumes that port inputs are asynchronous within a cycle and that port outputs update the port register synchronously at the end of the cycle. Thus, the `port_ready` input is ignored.

Architecture for the Unpipelined Single-cycle Organization

The file *pP-unpipelined_single_cycle_rtl.vhd* contains the architecture body for this organization of the pP. While the code is at the register-transfer level of abstraction, it does not conform to the standard synthesis guidelines. This is because of the need for asynchronous reads of the instruction and data memories and the GPR register file.

The signals declared within the architecture connect the various functional units in the organization of the pP. The units are represented by the processes and assignment statements in the architecture body. Aliases are used to refer to fields of the current instruction word represented by the `IR` signal.

The first group of processes and assignments deal with control flow in the pP. The assignment to `branch_taken` uses the branch function code from the current instruction to determine whether the branch is taken. The assignment to `incr_PC` determines the next sequential instruction address. The assignment to `next_PC` determines the instruction address for the next instruction to be executed:

- If an interrupt request is pending, the next PC is 1.
- If the current instruction is `RETI`, the next PC is taken from the interrupt register (`int_PC`).
- If the current instruction is `RET`, the next PC comes from the return-address stack.
- If the current instruction is `JMP` or `JSB`, the next PC is the target address.
- If the current instruction is a taken branch, the next PC is the sum of the incremented current PC and the displacement.
- Otherwise, the next PC is the incremented current PC.

The `PC_reg` process represents the synchronous storage for the PC. It resets the PC to zero when the system is reset, and updates the PC using the calculated next PC value at other times.

The `int_reg` process represents the synchronous storage for the interrupt register, including the saved PC (`int_PC`) and condition code bits (`int_z` and `int_c`) and the interrupt-enable bit (`int_en`). It also controls the `int_ack` port. On system reset, interrupts are disabled and `int_ack` is negated. When an interrupt request is pending, the interrupt register signals are updated, interrupts are disabled and `int_ack` is asserted. Otherwise, `int_ack` is negated and, if the current instruction is `ENAI` or `DISI`, the interrupt enable bit is updated accordingly.

The `instr_mem` process represents the storage for the instruction memory. The constant `IMem` is initialized using the `load_program` procedure. Whenever the current PC value changes, the process fetches the instruction at the new PC address and assigns it to the `IR` signal, representing the current instruction to be executed.

The `stack_mem` process represents the storage for the return-address stack and stack pointer. When the system is reset, the stack pointer is cleared to zero. At other times, when no interrupt request is pending, the process checks the current instruction. If it is a `JSB`, the process pushes the incremented PC (the address of the instruction after the `JSB`) and increments the stack pointer. Alternatively, if the instruction is a `RET`, the process decrements the stack pointer to pop the stack. In all cases, the process assigns the top value on the stack to the `stack_top` signal, representing the current return address.

The `GPR_mem` process represents the general purpose register file. The first part of the process deals with synchronously updating register contents at the end of a clock cycle. On system reset, all registers are cleared to zero. On other cycles where an interrupt is not pending and the destination register is not r0, the process stores the write data in the register file at the address given by `IR_rd`. The source of data depends on the current instruction. The second part of the process deals with asynchronously reading register contents for the two read ports. One read port simply uses the address on `IR_r1` to access the register file. The other read port uses `IR_rd` for store and output instructions; otherwise it used `IR_r2`. Since the process deals with both synchronous and asynchronous operation, it includes in its sensitivity list all of the signals that it reads.

The `ALU` process represents the hardware that performs arithmetic, logical and shift operations. Implementation of the operations is the same as in the behavioral model. The opcode and function code fields of the current instruction are used to select the operand sources and the operation performed. The process assigns its data result to the `ALU_result` signal and its carry out result to the `ALU_c` signal. The zero condition code is calculated separately by the assignment to the `ALU_z` signal.

The `cc_reg` process represents the storage for the condition codes. They are cleared to zero on system reset and updated from the ALU condition codes when an ALU or shift instruction is executed with no interrupt pending.

The `data_mem` process represents the storage for the data memory. It is synchronously updated when a store instruction is executed with no interrupt pending. The data memory is asynchronously read using the ALU output as the effective address.

The next group of assignments deal with the I/O port interface for input and output instructions. In all cases, the signals are only activated if no interrupt is pending. The `port_addr` signal is driven with the effective address from the ALU output when an input or output instruction is executed. The `port_data` output is enabled with register data when an output instruction is executed and is tristated at other times. The `port_read` signal is activated for an input instruction, and the `port_write` signal is activated for an output instruction.

The final process issues debug messages. It is encapsulated in a conditional generate statement that only includes the process if the `debug` generic is true. The process issues messages upon system reset, upon acknowledgment of an interrupt and upon execution of each instruction.

An Unpipelined Multi-Cycle Organization

Figure 2 shows an unpipelined pP organization that takes multiple clock cycles to execute each instruction. On each cycle, one step of instruction interpretation is performed, and the machine state is updated at the end of the cycle. Different instructions may take different numbers of cycles, depending on the interpretation steps required. The advantage of this approach over the single-cycle approach is that much less work needs to be done per cycle, so the cycle time can be faster. Furthermore, many instructions do not require all interpretation steps, so their execution will be faster than for the single-cycle implementation.

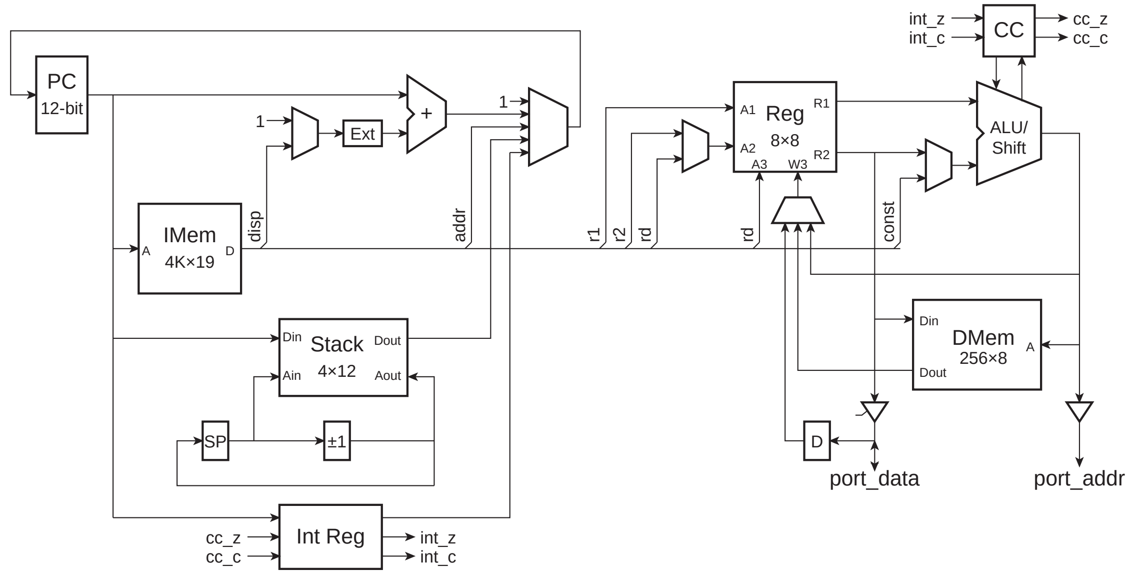

FIGURE 2 An unpipelined multi-cycle organization for the pP.

The first cycle of execution involves checking whether an interrupt request is pending. If one is, the current PC and condition code bits are saved in the interrupt register and the PC is set to 1. Execution of the interrupt service code then proceeds in the subsequent cycle.

If no interrupt is requested, the first cycle is used to index the instruction memory to fetch the instruction to be executed. In this implementation, the instruction memory is a synchronous ROM, and the ROM output register forms the instruction register (IR). The PC register is updated with the incremented PC value.

During the second cycle, the GPR register file is accessed to fetch operands, in case they are required. The register file in this implementation has synchronous read ports, and the operands are stored in two output registers. Also in this cycle, control-flow processing is performed. If the instruction in the IR is a conditional branch that is taken, the PC is updated with the sum of its current value and the branch displacement. If the instruction is a `JMP`, the PC is updated with the target address. If the instruction is a `JSB`, the PC is updated with the target address, the current PC value is pushed onto the return-address stack, and the stack pointer is incremented. If the instruction is a `RET`, the PC is updated from the top of the stack, and the stack pointer is decremented. If the instruction is a `RETI`, the PC and condition codes are restored from the interrupt register, and interrupts are enabled. If the instruction is an `ENAI` or `DISI`, the interrupt enable bit is set accordingly. In all cases of control flow instructions, processing is complete after the second cycle.

The third cycle (if required) involves computation of a data result or an effective address by the ALU. The result is stored in an output register. Also, for arithmetic, logic and shift instructions, the condition code bits are updated.

For memory and I/O instructions, a further cycle is used to access the memory or port register. The ALU output register is used as the address. The data memory in this implementation reads and writes synchronously. For memory stores, write data from the GPR register file output register is stored at the end of the clock cycle. For memory loads, read data is made available at the data memory output register at the end of the cycle. For port input and output instructions, the pP checks the `port_ready` input at the end of the cycle. If it is negated, the pP repeats the cycle, allowing the port controller extra time to read or write the data. When `port_ready` is active, the input or output operation is complete. For input instructions, the port data is stored in the data input register.

A final cycle is required for instructions that update a destination register in the GPR register file, namely, arithmetic, logic, shift, load and input instructions. The data source is one of the ALU output, the data memory output or the port input data register, depending on the instruction. The destination register (if not r0) is updated at the end of the cycle.

Architecture for the Unpipelined Multi-cycle Organization

The file *pP-unpipelined_multi_cycle_rtl.vhd* contains the architecture body for this organization of the pP. The code is largely similar to that for the single-cycle implementation. The main difference is that the architecture includes a state machine to sequence execution over several cycles. Other differences arise from all of the storage having synchronous outputs.

The state machine is implemented by the two processes `control` and `state_reg`. `Control` determines the state for the next cycle based on the current state, the interrupt enable and request inputs and the fetched instruction opcode and function codes. In the case of input and output instructions, the state does not advance beyond `mem_state` until the `port_ready` input is asserted.

In this implementation, selection of the next PC value is folded into the `PC_reg` process. The PC is reset to zero on system reset. In `fetch_state`, the PC is either set to 1 or incremented, depending on whether there is a pending interrupt. In `decode_state`, the PC is updated, if required, for a control flow instruction.

The `int_reg` process is similar to the previous implementation, except that update of the interrupt enable flag for `RETI`, `ENAI` and `DISI` instructions is deferred to `decode_state`. The `instr_mem` process is also similar to the previous implementation, except that instruction memory is read synchronously during `fetch_state`.

The `stack_mem` process is somewhat different, due to the multi-cycle timing. The stack is updated during `decode_state`. For a `JSB` instruction, the PC has already been incremented by `decode_state`, so the value from the PC register is saved rather than the incremented value. The `GPR_mem` process is also different. The destination register is updated during `write_back_state`, and operand registers are read during `decode_state`.

The `ALU` process and `ALU_z` assignments are the same as in the single-cycle implementation. However, since the combinational ALU result needs to be stored, the `ALU_reg` processes is added. It stores the result at the end of `execute_state`. The `cc_reg` process is modified to store the condition code bits at the end of `execute_state` also.

The `data_mem` process is modified to perform its operation in `mem_state`. The output of the data memory is synchronous. The I/O port assignments are also modified so that they are only active during the `mem_state`. The `port_reg` process is added to store the input data value at the end of `mem_state` for input instructions.

Finally, the debug monitor process is modified so that the PC is captured during `fetch_state`, but the trace message for instruction execution is not written until `decode_state`. This is because the fetched instruction is not available on the `IR` signal until `decode_state`.

Projects

1. Develop a parallel port controller as an input/output device and interface it with the pP.
2. Synthesize and implement the multi-cycle architecture for an FPGA, and download it to an FPGA development board.
3. Develop a pipelined architecture for the pP.