CAD6 Program Counter

Assignment

To design the program counter (PC) for your microprocessor. Make this circuit scannable for inclusion in a scan chain.

Description

PC

All instructions start by using the contents of the program counter as the address to fetch the next instruction. The program counter stores the current instruction address and calculates the address of the next instruction. Storing the current instruction address requires nothing more than a resetable 16-bit register. However, depending on the current instruction, we must also determine whether the next instruction is the instruction that follows sequentially or the instruction resulting from a jump or a branch.

Except on Jumps and Branches, the next instruction follows sequentially from the current instruction (i.e., PC ← PC + 1). The following table lists the instructions from the baseline architecture which may force an address different from the next sequential address in the PC.

Table 3: PC Modifying Instructions

INSTRUCTION	NEXT PC VALUE
Bcond disp	PC ← PC + disp (sign ext.) or PC ← PC + 1
Jcond Rdest	PC ← Rdest or PC ← PC + 1
JAL Rlink, Rdest	Rlink ← PC + 1 PC ← Rdest

To avoid confusion concerning the value of the displacement amount for branches, consider the output listing below from the assembler. Note that the displacement for the ble at 0x141 is 0x0F (e.g. 0x142 + 0xF = 0x151), not 0x10! Defining the displacement in this manner makes the design of your program counter simpler since the pc has been incremented to 0x142 by the time you're ready to calculate the new branch address.

```
0140 / 0020;    #0000000000100000  ( 99) test5          or     r0 r0
0141 / C70F;    #1100011100001111  (100)                ble    j1
0142 / 0434;    #0000010000110100  (101)                xor    r4 r4
                #0000000101010001  (103)                .orig  0x0151
0151 / 0020;    #0000000000100000  (104) j1             or     r0 r0
0152 / CC0F;    #1100110000001111  (105)                blt    j2
0153 / 0434;    #0000010000110100  (106)                xor    r4 r4
```

A complication is presented by the fact that the processor is pipelined. One must decide how to handle changes in program flow in general, as the change in program flow happens, in our case, in the second stage. Meanwhile, unless we design the control to do differently, the processor would continue to fetch and decode subsequent instructions that follow the branch or jump. Several possible solutions are:

1. Always follow a branch or jump instruction with a NOP, or with other instructions that should be executed anyway. This is done by the compiler in many RISC processors. This is the simplest solution from a hardware point of view. Often the branch delay slots can be filled with useful instructions.

2. Lock the first stages of the pipeline so that they do not continue to fetch and decode instructions following a jump or branch. This is equivalent to forcing NOPs in the hardware. This approach adds some complexity, while reducing code size somewhat, and reducing throughput somewhat.

3. For conditional branches, guess whether program flow will be changed or not, and continue fetching and decoding instructions. This requires a fast decode (to know whether the instruction is a conditional branch or jump), a separate arithmetic unit to calculate PC values, and the ability to squash instructions in the pipe if the guess was wrong (this is found out when the instruction gets to the second stage).

Option 1 is acceptable for the baseline machine.

You should be able to initialize the register to a known state, though this state may not necessarily be 0x0000, depending on your application. For example, sometimes processors have operating modes such that they come out of reset fetching from internal memory in one mode and from external memory in another mode.

Your program counter must be made scannable in an effort to improve the testability of your design. You can decide later whether you'll implement a single scan chain or multiple scan chains.

Implementation

Register Implementation

From Table 1 it is clear that the next PC value to be stored could be from either an incrementer (normal), the register file (Jump), or an adder (Branch). The PC unit block shown in your baseline architecture block diagram, is assumed to include an adder. Branch and Jump instructions could alternatively use the ALU to calculate the next address. It would also be possible to use the ALU to increment the PC in the normal case, too, thereby reducing chip area, but complicating control and routing, and lengthening the critical path. The method implied by the bus interconnections in the baseline architecture block diagram is shown below in Figure 1.

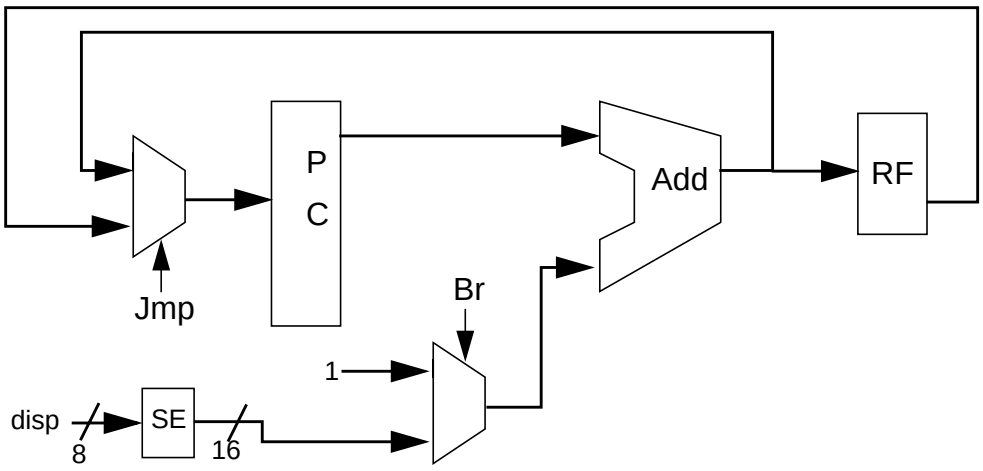

Figure 1: One Method of PC Implementation

Synchronous Counter Implementation

You may design the PC/incrementer as a 16-bit **synchronous** up-counter with reset and parallel loading capability. In the common case, it will auto-increment every clock cycle. The parallel loading ability would be used for loading the branch/jump target calculated by the ALU.

Counters are one of the simplest types of sequential circuits. A counter is usually constructed from two or more flip-flops which change state in a prescribed sequence when input pulses are received. Synchronous counters have an advantage over asynchronous counters in that the stages are clocked simultaneously and the outputs change in a synchronous manner.

In the circuit below, the first stage operates as a simple divide by two stage, with Q' fed back to the D input. Subsequent stages drive Q or Q' back to D via a multiplexer. (The register bits hold their state when Q is fed back and change state when Q' is fed back.) This change of state is enabled when all of the previous stage Q signals are true.

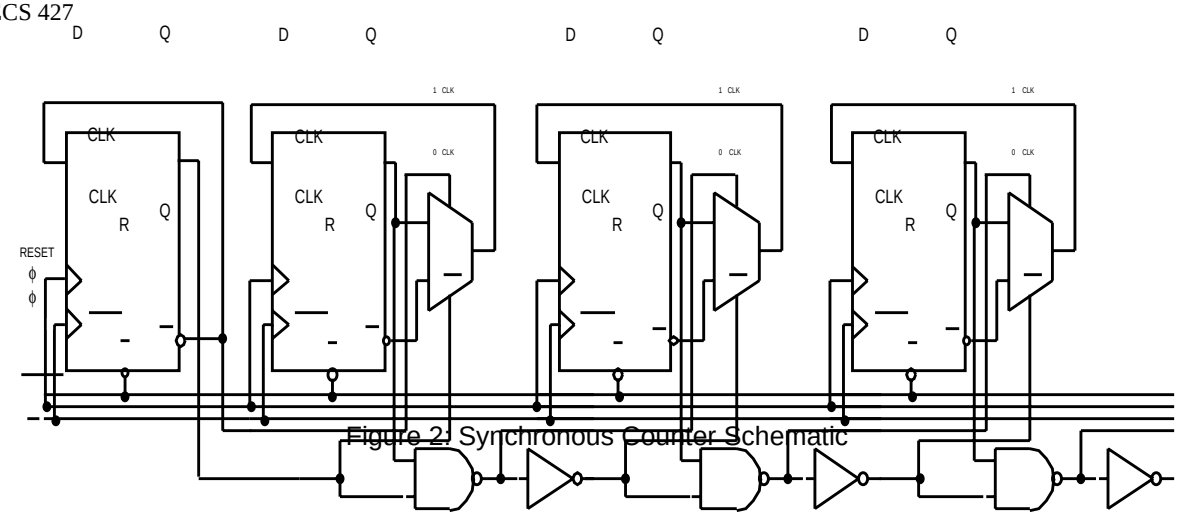

Figure 2: Synchronous Counter Schematic

Comments

Your PC circuitry must properly execute the instructions in the baseline architecture. Be sure that it will function properly in the pipelined environment. If you add interrupt capability to your processor, you must make appropriate extensions to the PC unit. Be sure that you can easily reset your PC, or set it to some predetermined state.

The PC contents are important state information which you will want to be able to control and observe when testing your processor. Therefore, these should all be implemented in scannable flip-flops or latches.

You can save routing work and chip area by putting the PC on the datapath. **Your scannable registers should be pitch-matched or bitslice-width matched to the datapath.** By using the bus wires already in place, you will save chip area.

Requirements

Please turn in the following in your **cad6** directory:

- A README file describing the approaches you use to implement the PC and why you used those approaches. Describe how branch targets are calculated (in PC or in ALU?). As usual, you may add any other comments you think are pertinent.
- Schematics for the Program Counter.
- Modelsim .wlf/.do, with delays added, for the PC showing all the different possibilities for next address calculation implemented in your processor.
- Layout of the PC.
- Eldo config files showing delays. Describe how you arrived at the delays, referencing any schematics you may have created for the purpose of delay calculation.
- DRC, LVS and PEX reports for the PC.