CAD6 Program Counter Winter 2009

Assignment

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

Description

Program Counter

All instructions start by using the contents of the program counter (PC) as the address to fetch the next instruction. The PC stores the current instruction address and calculates the address of the next sequential instruction. Storing the current instruction address requires nothing more than a resettable, 16-bit register. Except on Jumps and Branches, the next instruction follows sequentially from the current instruction (i.e., PC ← PC + 1). On branches and jumps you should take the new PC address as an input from the datapath. Additionally, the PC needs to be able to stall (keep the same value) for instruction cache misses. When designing deeper pipelines, you may want to compute the branch target address in the PC to save one cycle delay for branches, but this is optional for our 2-stage baseline processor. 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 1: 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 arises due to the fact that the processor is pipelined. One must decide how to handle changes in program flow, in general. In our case, the change in program flow occurs in the second stage. Meanwhile, unless we design the control differently, the processor will 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 and is the simplest solution from a hardware point of view. Often, these 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 that the branch will not be taken and continue fetching and decoding instructions; then squash mis-predicted instructions. This is the most rudimentary type of branch prediction and is only slightly more complicated than (2) but it loses less throughput with the same code size advantage.
4. For conditional branches, guess whether the branch will be taken or not and squash if wrong. This is significantly more complicated and it requires a fast decode (to know whether the instruction is a conditional branch or jump), a separate arithmetic unit to calculate PC values, and everything required for (3), but it has the potential of being the fastest.

Option 1 is acceptable for the baseline machine. So, what does the PC need to do? You should be able to initialize the register to a known state, though this state may not necessarily be x0000, 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. It must also increment, load in a value for jumps and either load in a value from the ALU or calculate and load a target address for branches. Your program counter must also 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. The scan means that your PC must have another mode where it acts as a shift register with a one bit shift in and a one bit shift out.

You should layout the program counter appropriately for inclusion into your datapath (i.e., inputs and outputs should be arranged in a bit-sliced manner that will allow for easy integration with the datapath).

Implementation

Register Implementation

From Table 1 it is clear that the next stored PC value could come from a variety of places: it could come from a counter that increments by 1 (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 connections in the baseline architecture block diagram is shown below in Figure 1.

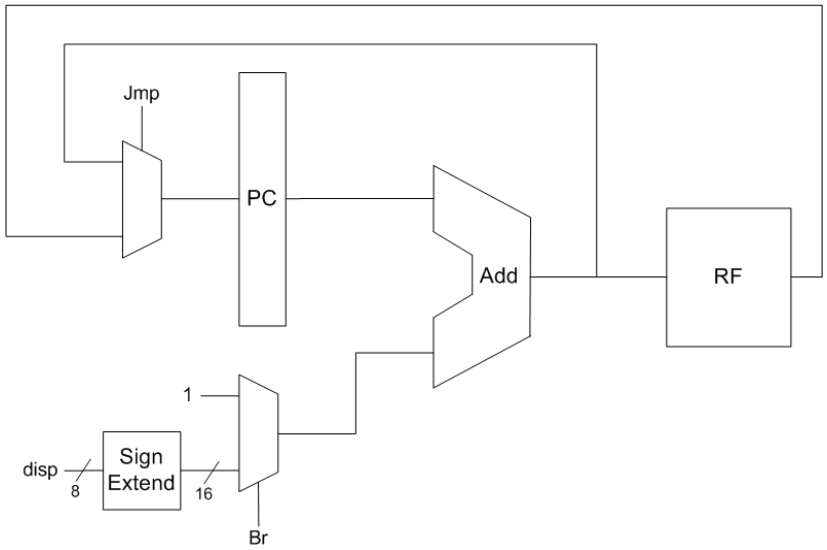

Figure 1: One method of PC implementation.

Synchronous Counter Implementation

You may also 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.

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.

Remember you are working with a lot of new tools. Plan your time ahead and take advantage of the office hours for help with Synthesis/APR tools.

You can use the verilog file */afs/umich.edu/calss/eecs427/w09/resource/ROVSD16x1024.v* as your off-chip Instruction ROM.

Requirements

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

- A README file describing the approaches you used 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.
- Behavioral Verilog for the PC.

- Design Compiler scripts for the PC (justify all synthesis timing constraints in your README file).
- Encounter APR scripts for the PC.
- Structural Verilog for the PC (output of APR).
- Layouts for the PC which passes DRC and LVS.
- Synthesis Timing Reports for the PC (describe the worst case path in the README).

Deadline

You need to turn in CAD 6 by **Tuesday, March 10, 2009, 7pm**.