CONCORDIA UNIVERSITY
Department of Electrical and Computer Engineering
COEN 316 Computer Architecture
Lab 3
Next-address Unit
Fall 2021

The Next-Address unit is responsible for generating the next address which is to be stored in the Program Counter (PC) register. Before describing the functionality of the next-address unit, we will first examine encoding format of the jump and branch instructions. The five instructions (expressed in assembly language) are:

```
jump there      ; jump to memory location "there"
jr   rs         ; jump to memory location whose address is in rs
beq rs,rt, loop ; jump to memory location "loop" if rs=rt
bne rs,rt, loop ; jump to memory location "loop" if rs /= rt
bltz rs, loop   ; jump to memory location  "loop" if rs < 0
```

All instructions are encoded in a 32 bit wide format with 6 bits used to represent the opcode. Memory addresses are 32 bits wide; consequently the jump instruction uses a variation of direct addressing known as "pseudo-direct" addressing since it is not possible to store the entire 32 bit address within a jump instruction. **For the version of the processor implemented in these labs, a simpler form of this pseudo-direct addressing will be used**. Figure 1 gives the instruction format for the unconditional jump and the three conditional branch instructions.

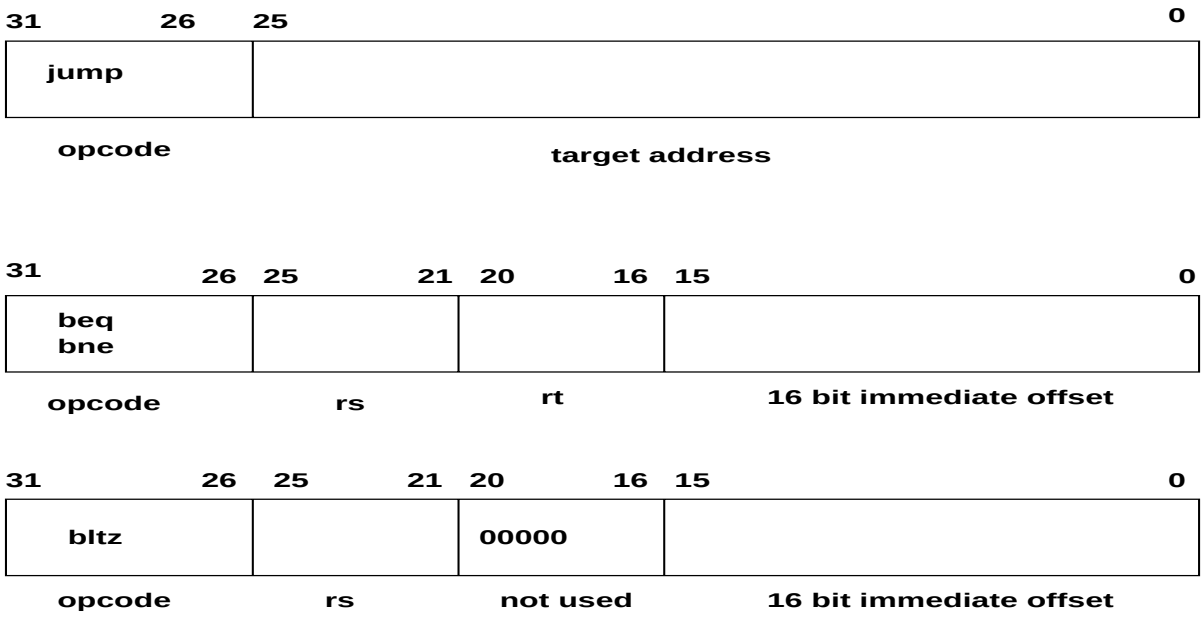

Figure 1: jump and branch instruction formats.

For a jump instruction, the low order 26 bits of the instruction (bit 25 downto bit 0) are padded with 6 0s to form a full 32-bit address which is the target address of the jump instruction. For example, consider the following assembly language program:

```
memory address  label     instruction

00000                     addi r1,r0, 1  ; r1 = r0 + 1
00001                     addi r2,r0, 2  ; r2 = r0 + 2
00010           there:    add  r2,r2, r1 ; r2 = r2 + r1
00011                     jump there
00100                     "next instruction"
```

When assembled into actual machine code, the symbolic label "there" would be replaced with the 26 bit value of 00000000000000000000000010 which is stored as part of the actual machine code instruction.

For the conditional branch instructions, the 16-bit immediate data is sign extended to form a 32 bit offset which is added to the value of PC (the value of PC after the branch instruction has been fetched) to form a branch target address. Hence, branches are relative to PC and jump instructions are absolute (or pseudo absolute to be precise). The following program illustrates the relative offset of a conditional branch instruction:

```
memory address  label     instruction

00000                     addi r3,r0,6 ; r3 = r0 + 6
00001                     addi r1,r0,0 ; r0 = r0 + 0
00010                     addi r2,r0,1 ; r2 = r0 + 1
00011           loop:     add r1,r1,r2 ; r1 = r1 + r2
00100                     addi r2,r2,1 ; r2 = r2 + 1
00101                     bne r2,r3 loop
00110                     "next instruction"
```

When assembled into machine code, the "loop" label would be replaced with the relative offset of -3 (stored as a 16 bit signed value in two's complement representation) since the value of PC = 6 after the bne instruction has been fetched (adding the branch target offset of -3 with the value of PC = 6 yields the target address of 3).

The next address to be loaded into the PC register is generated in one of four methods:

- PC = PC + 1 no branching, straight-line execution
- PC = 000000: 26 bit jump target address in the case of a jump instruction
- PC = PC + 1 + signed extended 16 bit immediate target address in the case of a branch instruction and the branch condition is satisfied.
- PC = contents of register rs in the case of a jump register (jr) instruction

Figure 2 gives the “black box” diagram of the next-address unit.

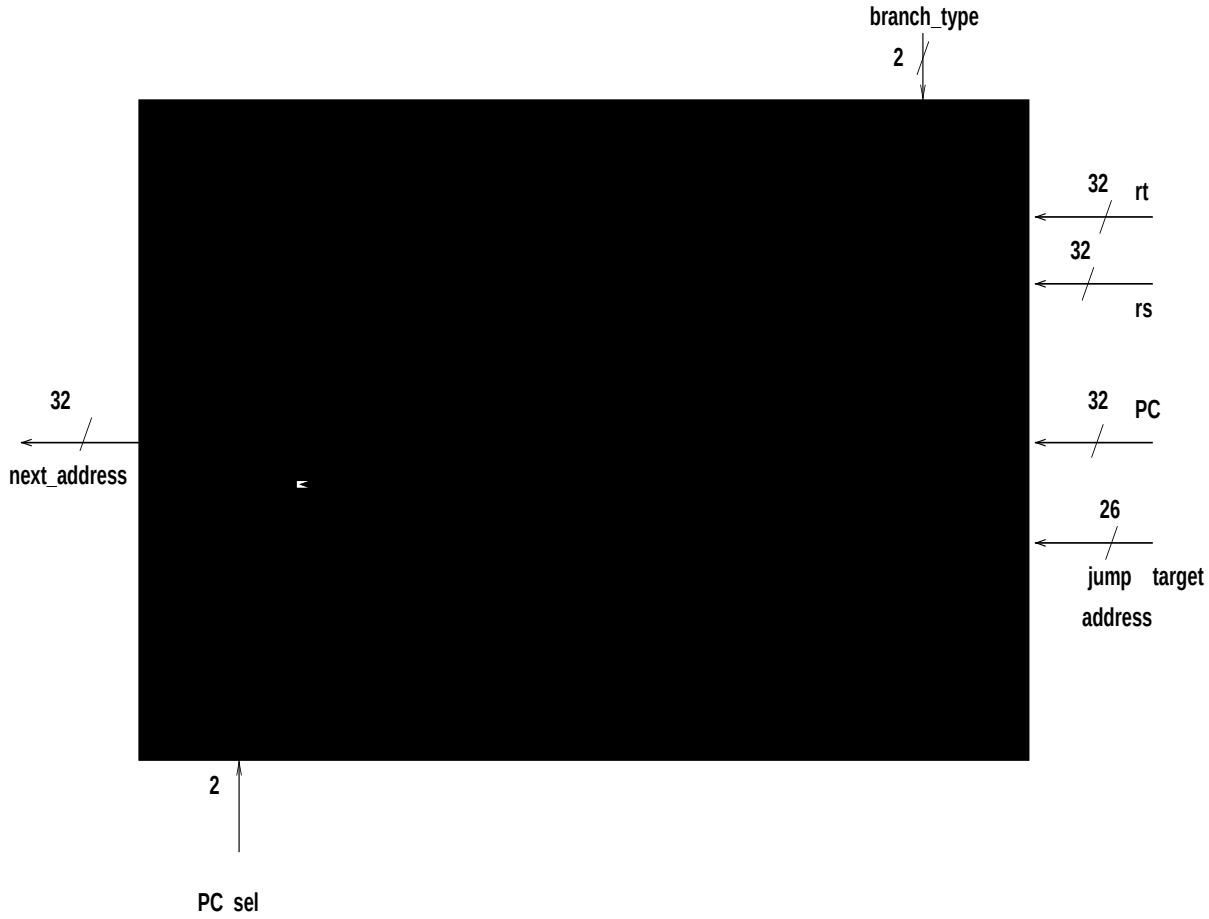

Figure 2: Next-address unit “black box” view.

The inputs to the next-address unit are:

• the 32 bit value of the PC register

• the contents of the rs and rt registers (which need to be compared in the case of branch instructions.

• the 26 bit jump target address stored within a jump instruction (in the case of a branch instruction the low order 16 bits of these 26 bits form the 16 bit immediate data representing the signed branch offset value which is to be sign extended to a 32 bit signed offset)

• a 2 bit PC_sel (which will be generated by the control unit) which selects one of the three inputs to a multiplexer according to the following table:

Table 1: PC_sel functionality

PC_sel	comment
00	no *unconditional* jump (**straightline** instructions as well as the **conditional** branches will have PC_sel = 00).
01	jump
10	jump register
11	not used

As shown by Table 1, the control signal PC_sel will be set (by the control unit) to 00 **for all the "non jump" instructions** (such as the arithmetic and logical instructions as well as the 3 conditional branch instructions (`beq`, `bne`, `bltz`)). In the case of a `jump` instruction (an unconditional jump) the value of PC_sel will be set to 01 by the control unit. Finally, a `jump register` instruction will be selected with PC_sel set to (by the control unit) to 10.

• a 2 bit branch_type (also generated by the control unit) which specifies one of four possible branch types (note that PC_sel will be = 00 for all the 4 possible values of the branch_type signal) :

Table 2: branch_type functionality

branch_type	Meaning	Value to be added to PC
00	no branch (straight-line code such as `add`, `sub`, `and`, `xor`, etcetera)	1 (as a 32 bit number)
01	`beq` (*conditional* branch equal to 0)	1 + branch offset value sign extended to 32 bits if rs = rt, otherwise 1 is to be added
10	bne (*conditional* branch not equal to 0)	1 + branch offset value sign extended to 32 bits if rs /= rt, otherwise 1 is to be added

Table 2: branch_type functionality

branch_type	Meaning	Value to be added to PC
11	`bltz` (*conditional* branch less than zero)	1 + branch offset value sign extended to 32 bits is rs <0, otherwise 1 is to be added

As indicated in Table 2, the 2 bit branch_type input determines the type of branching and thus also determines the value which is to be added to the PC register in order to form the next address. If the branch_type is 00 indicating that **no branching** is to be performed, the constant value 1 is to be added to PC (implying straight line execution). For the remaining three values which specify the conditional branching instructions (branch equal, branch not equal, branch less than zero), if the condition is satisfied (as determined by comparing the contents of the rs and rt registers) the value of 1 + sign extended branch offset value is to be added to PC, otherwise the value of 1 is added to PC if the condition is false.

The careful reader may have noted some differences between the details of the implementation of the next-address unit as presented in this lab handout and the actual implementation of the processor as described in the textbook. The notable differences (which were introduced in the interest of simplifying the design) are:

• main memory will consist of locations with each addressable location consisting of 32 bits (as opposed to a byte addressable memory which would require 4 bytes per 32bit word).

• as a consequence of the above main memory model, the value of the PC register is incremented by 1 (instead of 4) during straightline execution, or by 1 + target address in the case of the conditional branch (when the condition is satisfied).

• the method of converting the 26 bit “pseudo-direct” address stored as part of the instruction in the unconditional jump instruction has been simplified to simply padding 6 bits of 0 to the left (as opposed to multiplying the address field by 2 and augmenting the 4 high-order bit of the PC register to the stored address field).

Procedure

Before writing any VHDL code, you should design the next-address unit at the register-transfer level (RTL) in block diagram form by using Figure 2 as a starting point. Your task is to fill in the details of the black-box based upon the previous description of the functionality of the next-address unit. At the register-transfer level, design is performed using components such as multiplexers, adders, registers, etc. We are not concerned with the actual gate-level implementation of these units. It is quite apparent that minimally a parallel adder and a multiplexer will be required. You

are to determine what the inputs to the adder are to be (HINT: one of the adder inputs is the PC register). You are also to determine the type of multiplexer and what are the inputs to it, and you are to determine whether any other components are required in order to complete the design of the next address unit.

Design the next address unit using VHDL and the following entity specification:

```
entity  next_address   is
port(rt, rs : in std_logic_vector(31 downto 0);
              -- two register inputs
     pc     : in std_logic_vector(31 downto 0);
     target_address : in std_logic_vector(25 downto 0);
     branch_type    : in std_logic_vector(1 downto 0);
     pc_sel         : in std_logic_vector(1 downto 0);
     next_pc        : out std_logic_vector(31 downto 0));
end next_address ;
```

You may use any style of synthesizeable VHDL (i.e. structural with port maps, processes, CSA statements, etc.) you wish. Simulate your design with the Modelsim simulator to verify correct functioning for typical input values. Ensure your simulation fully tests the cases of no branch (straight line code), `beq`, `bne`, `bltz`, as well as the two `jump` instructions. Synthesize your VHDL code with Xilinx Vivado to generate a .bit file.

Board Implementation

Similar to the two previous labs, only a subset of the various input and output ports will be mapped to switches and LEDs via a .xdc file. You are to simulate the full 32-bit design and to create a "board wrapper" VHDL file for implementation. Use the following entity specification for the "board wrapper" VHDL code:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity  next_address   is
port(rt_in, rs_in      : in std_logic_vector(1 downto 0);
     pc_in             : in std_logic_vector(2 downto 0);
     target_address_in : in std_logic_vector(2 downto 0);
     branch_type       : in std_logic_vector(1 downto 0);
     pc_sel            : in std_logic_vector(1 downto 0);
     next_pc_out       : out std_logic_vector(2 downto 0));
end next_address ;
```

Requirements

1. Modelsim simulation results for the (full 32-bit) design.

2. The VHDL code for both the full 32-bit version and the ‘board-wrapper’ version.

3. The Vivado `runme.log` log files for the synthesis and implementation . Discuss the importance of any messages or warnings.

4. Demonstrate the operation of your next-address unit on the FPGA board to your lab TA (you may demo at the beginning (first 30 minutes) of your next lab session).

References

1. Computer Architecture From Microprocessors to Supercomputers, Behrooz Parhami, Oxford University Press, ISBN 0-19-515455-X, 2006

Revised: T. Obuchowicz, October 12, 2017. Modified to allow for more design work on part of the designer, i.e. the students.

Revised: T. Obuchowicz,October 14, 2021. Revised for use with Xilinx Vivado GUI and Nexys A7 FPGA board.

Ted Obuchowicz
October 19, 2016