ASIP Laboratory - Session 4

Paul Georg Wagner Majd Mansour

June 3, 2017

Exercise 1: Adding a new instruction to *dlxsim*

Adding a new instruction to dlxsim takes several steps. First of all we defined a new unique index for the new instruction `bgeu` and added it as `OP_BGEU` to the file *dlx.h*. Then we inserted the new operation at the respective index of the `operationNames` array that is defined in *sim.c*.

In the next step the instruction format has to be defined. The instruction `bgeu r1, r2, label` branches to a label if the first operand register is greater than or equal to the second operand register. This instruction format is resembled by the `BRANCH_2_OP` macro in *asm.c*. The opcode for the new instruction was uniquely specified as `0xB0000000`. Hence we inserted the struct `{"bgeu",BRANCH_2_OP, 0xB0000000,0xfc000000,0,0,0}` to the end of the `opcodes` array in *asm.c*. Finally – since we used the most significant six bits for the opcode – the new operation also had to be included into the `opTable` array in *sim.c*.

The last step of adding a new instruction to dlxsim is to implement the operation semantics in the `Simulate()` function in *sim.c*. For this we copied and modified the case block of the already existing `OP_BEQZ` instruction, which is very similar to the new instruction. The main difference of the two instructions is that `beqz` only uses one operand register, while `bgeu` uses two. In order to accommodate this, we added a call to the `LoadRegisterD` macro alongside the already existing `LoadRegisterS1` macro call. This results in the two variables `rs1` and `rd` being initialized with the respective operand values. Since `bgeu` uses the I-instruction format (6 bits opcode, 5 bits register, 5 bits register, 16 bits immediate), which is usually used for immediate instructions like `addui`, the destination register `rd` had to be used for this purpose. The more appropriately named `rs2` variable is only available with the R-instruction format (e.g. `add` instruction), which does not feature the necessary immediate field for the jump label.

Besides the loading of the additional input operand, the condition for the branch was implemented using a simple if-statement comparing `rs1` and `rd`. Since the `bgeu` instruction works on unsigned values, the signed integer variables `rs1` and `rd` had to be casted to unsigned integers prior to comparison. Otherwise this can lead to problems when very large unsigned integers are compared. In that case the C code interprets the big unsigned integer (MSB is set to 1) from the input registers as negative integers, which breaks the correct semantics of the comparison. The code that performs the actual branch by setting the program counter has been adopted from the `beqz` case. Listing 1 displays the case-statement for the newly implemented `bgeu` instruction.

Listing 1: New case statement in *sim.c*

```c
case OP_BGEU:
    LoadRegisterS1 LoadRegisterD
    if ((unsigned int)rs1 >= (unsigned int)rd) {
        pc = readRegister(machPtr, NEXT_PC_REG, 0)
              + ADDR_TO_INDEX(wordPtr->extra);
        machPtr->branchYes++;
        /* ASIP JUMP : */
```

```
            additionalClockCycleForJump = 2;
            if (machPtr->debugAssembly)
                memcpy(symGetString_Backup, Sym_GetString(machPtr,
                    INDEX_TO_ADDR(last_pc)), 100*sizeof(char));
            addBranchToTraceFile(machPtr->traceJumpsFile,
                INDEX_TO_ADDR(last_pc), INDEX_TO_ADDR(pc));
        } else machPtr->branchNo++;
        machPtr->branchSerial = machPtr->insCount;
        machPtr->branchPC = INDEX_TO_ADDR(readRegister(machPtr,PC_REG,0));
        break;
```

After the implementation of the new bgeu instruction has been completed, we first verified the correct implementation by running a very simple test program. This program performs a single comparison and – depending on the outcome – stores different values in registers. Using the test program we confirmed that the bgeu instruction works and also handles negative immediate values as desired.

a) *How many cycles do you need for execution? Attach bs_bgeu.s to the mail.*

   As soon as the new instruction was implemented we copied our bubble sort implementation from the last exercise and replaced the sltu and beqz instruction pairs with the new instruction bgeu. As expected, the resulting array was still correctly sorted. The updated bubble sort algorithm took a total of 5455 instruction cycles to execute.

   The new version is attached as *bs_bgeu.s*.

b) *What is the speedup compared to "bs_basis.s"?*

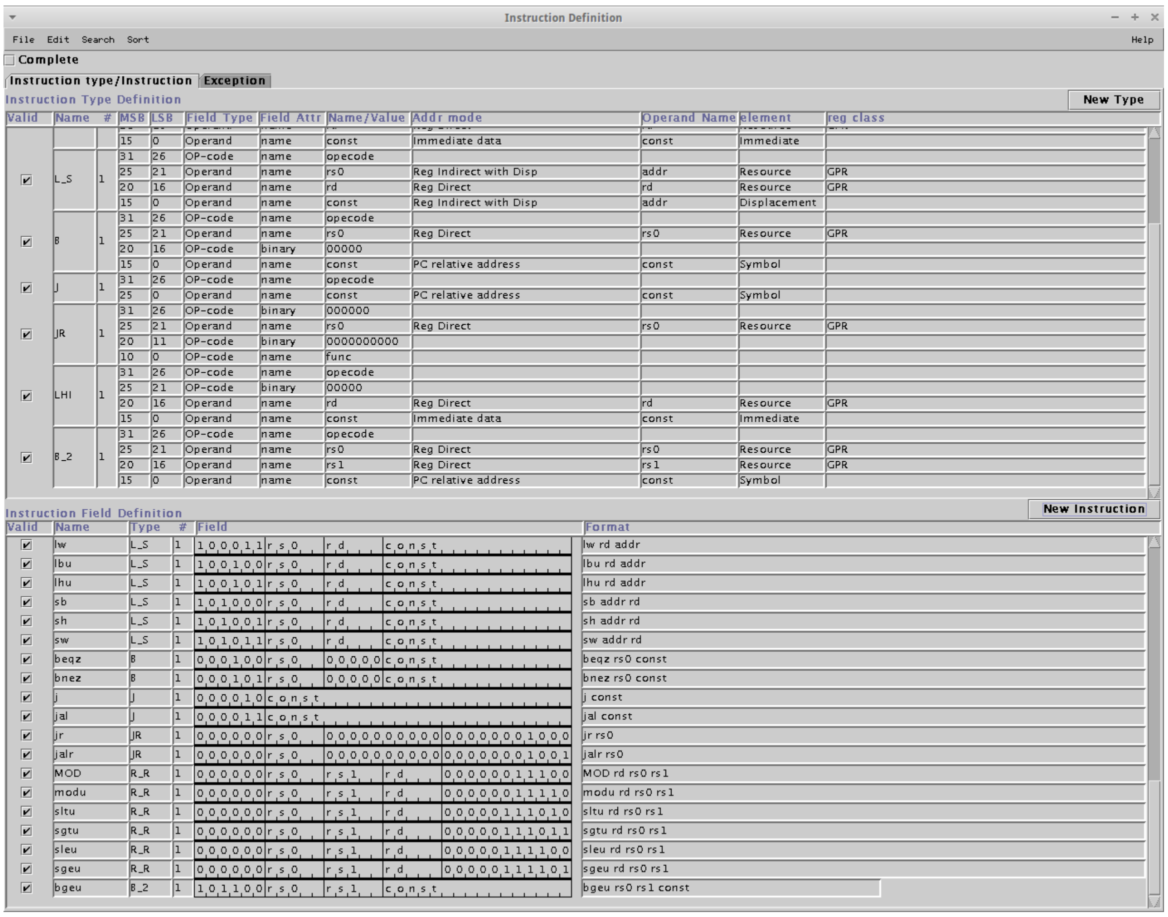
   The previous bubble sort version took a total of 6941 instruction cycles to execute. Since the new implementation only needs 5455 cycles, the modification results in a speedup of $\frac{6941}{5455} \approx 1,27$.

Exercise 2: Adding the new instruction to ASIPMeister

Adding the newly implemented instruction to the actual CPU description takes several steps as well. First of all we had to add a new instruction definition in ASIPMeister. This required the definition of a new instruction type, which holds 6 bits of opcode, two times 5 bits for input registers, as well as 16 bits for an immediate value. The immediate value then holds the PC-relative address of the jump target, while the two specified registers hold the values to compare. Afterwards we added a new instruction called bgeu of this instruction type. As opcode for this instruction the first six bits of 0xB0000000 are used, i.e. 101100. The new instruction format B_2 and the new instruction definition bgeu can be seen in figure 1.

As the second step the operands and the behavior of the new instruction has to be specified in ASIPMeister. The operands of the instruction are simply the two input registers (rs0 and rs1), the jump label (const) as well as the program counter (PC), as it has to be modified if the branch is taken. The behavior is specified by using a high-level if-statement. The result is shown in figure 2.

Finally the new instruction has to be implemented on the hardware level. For this, the pipeline structure of the processor has to be considered. In the IF-phase the instruction has to be fetched using the FETCH() macro. In the ID-phase the two input operands have to be fetched using the GPR2READ(rs0,rs1) macro. In the EXE-phase the actual instruction semantics have to be implemented. The bgeu instruction requires two main steps in the execution phase. First of all we have to use the cmpu function of the ALU to compare the two input values specified in rs0 and rs1. The ALU then subtracts the input operands from one another and sets several flags afterwards. We then have to check the signed flag, which



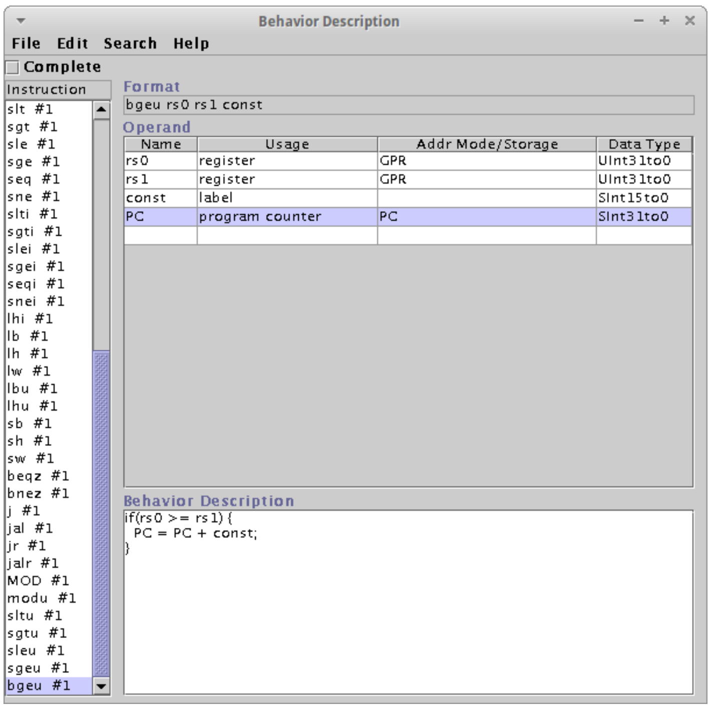
Instruction Type Definition

Valid	Name	#	MSB	LSB	Field Type	Field Attr	Name/Value	Addr mode	Operand Name	element	reg class
			15	0	Operand	name	const	Immediate data	const	Immediate	
✓	L_S	1	31	26	OP-code	name	opecode				
			25	21	Operand	name	rs0	Reg Indirect with Disp	addr	Resource	GPR
			20	16	Operand	name	rd	Reg Direct	rd	Resource	GPR
			15	0	Operand	name	const	Reg Indirect with Disp	addr	Displacement	
✓	B	1	31	26	OP-code	name	opecode				
			25	21	Operand	name	rs0	Reg Direct	rs0	Resource	GPR
			20	16	OP-code	binary	00000				
			15	0	Operand	name	const	PC relative address	const	Symbol	
✓	J	1	31	26	OP-code	name	opecode				
			25	0	Operand	name	const	PC relative address	const	Symbol	
✓	JR	1	31	26	OP-code	binary	000000				
			25	21	Operand	name	rs0	Reg Direct	rs0	Resource	GPR
			20	11	OP-code	binary	0000000000				
			10	0	OP-code	name	func				
✓	LHI	1	31	26	OP-code	name	opecode				
			25	21	OP-code	binary	00000				
			20	16	Operand	name	rd	Reg Direct	rd	Resource	GPR
			15	0	Operand	name	const	Immediate data	const	Immediate	
✓	B_2	1	31	26	OP-code	name	opecode				
			25	21	Operand	name	rs0	Reg Direct	rs0	Resource	GPR
			20	16	Operand	name	rs1	Reg Direct	rs1	Resource	GPR
			15	0	Operand	name	const	PC relative address	const	Symbol	

Instruction Field Definition

Valid	Name	Type	#	Field	Format
✓	lw	L_S	1	100011 rs0 rd const	lw rd addr
✓	lbu	L_S	1	100100 rs0 rd const	lbu rd addr
✓	lhu	L_S	1	100101 rs0 rd const	lhu rd addr
✓	sb	L_S	1	101000 rs0 rd const	sb addr rd
✓	sh	L_S	1	101001 rs0 rd const	sh addr rd
✓	sw	L_S	1	101011 rs0 rd const	sw addr rd
✓	beqz	B	1	000100 rs0 00000 const	beqz rs0 const
✓	bnez	B	1	000101 rs0 00000 const	bnez rs0 const
✓	j	J	1	000010 const	j const
✓	jal	J	1	000011 const	jal const
✓	jr	JR	1	000000 rs0 0000000000 00000001000	jr rs0
✓	jalr	JR	1	000000 rs0 0000000000 00000001001	jalr rs0
✓	MOD	R_R	1	000000 rs0 rs1 rd 00000011100	MOD rd rs0 rs1
✓	modu	R_R	1	000000 rs0 rs1 rd 00000011110	modu rd rs0 rs1
✓	sltu	R_R	1	000000 rs0 rs1 rd 00000111010	sltu rd rs0 rs1
✓	sgtu	R_R	1	000000 rs0 rs1 rd 00000111011	sgtu rd rs0 rs1
✓	sleu	R_R	1	000000 rs0 rs1 rd 00000111100	sleu rd rs0 rs1
✓	sgeu	R_R	1	000000 rs0 rs1 rd 00000111101	sgeu rd rs0 rs1
✓	bgeu	B_2	1	101100 rs0 rs1 const	bgeu rs0 rs1 const

Figure 1: The new instruction format and definition.



Format

bgeu rs0 rs1 const

Operand

Name	Usage	Addr Mode/Storage	Data Type
rs0	register	GPR	UInt31to0
rs1	register	GPR	UInt31to0
const	label		SInt15to0
PC	program counter	PC	SInt31to0

Behavior Description

```
if(rs0 >= rs1) {
  PC = PC + const;
}
```

Figure 2: The new instruction operands and behavior description.

specifies if the result was negative. If it is not set, the result of the subtraction was not negative, so rs0 is greater than or equal to rs1 and we have to perfom the jump. Otherwise, we do not change the program counter.

Before we change the program counter, the actual target address for the jump has to be calculated. This is also performed in this pipeline stage by adding the PC-relative immediate input value specified in const to the current value of the program counter. Since the ALU is already used for comparison in this stage, we specify a distinct carry-lookahead adder as additional hardware resource for this purpose. This adder ADD0 then calculates the jump target that PC is set to. Prior to this the already existing extender EXT0 resource is used for a sign extension on the const input value, since this 16 bit value can be negative (PC-relative addressing). The resulting 32 bit signed value then serves as input for the adder. The hardware description of the EXE-phase for the bgeu instruction is specified in listing 2.

Listing 2: EXE-phase of the bgeu instruction

```
wire [3:0]   flag;
wire         sign_flag;
wire         cond;
wire [31:0]  offset;
wire [31:0]  target;
wire         cin;
wire         cout;

flag = ALU0.cmpu(source0, source1);
sign_flag = flag[1];
cond = sign_flag == '0';

offset = EXT0.sign(const);
cin = '0';
<target,cout> = ADD0.adc(current_pc, offset, cin);
null = [cond] PC.write(target);
```

The modified CPU description then has been used in order to simulate the new bgeu instruction with ModelSim. For this, we modified our initial test program to also make memory writes depending on the outcome of the comparison. This allows us to check the correctness of the implementation by looking at the memory dump that ModelSim writes. We tested several different cases using our test program, which yielded the result that the CPU handles the comparison of positive values as desired. However, comparisons with negative values do not work equally well as they do with dlxsim. But since the bgeu instruction is defined explicitly for unsigned values only, this is not a problem.

Finally we also tested the bubble sort implementation with the new bgeu instruction in ModelSim. The final memory dump of ModelSim is shown in listing 3, which shows that the sorting algorithm still works very well with the new instruction.

Listing 3: Memory dump of the bubble sort program with bgeu instruction

```
00000000 00000004
00000004 00000006
00000008 00000007
0000000C 00000015
00000010 00000017
00000014 0000002A
00000018 0000002D
0000001C 00000036
00000020 0000003F
00000024 0000004B
```

```
00000028 0000004D
0000002C 0000004E
00000030 00000056
00000034 0000005C
00000038 00000063
0000003C 000000EB
00000040 00000156
00000044 0000015A
00000048 00000160
0000004C 0000027A
```

a) *Attach the ASIPMeister file for the "bgeu" CPU to the mail.*

The modified CPU description is attached as "dlx_basis.pdb".

Exercise 3: Optimizing the assembly code

In order to optimize the existing assembly code as much as possible, several measures can be taken. First of all, the complexity of the code can be reduced by avoiding to keep the two currently compared values updated in registers. This makes it possible to avoid the costly three-way swap, because the values can just be stored in memory in reversed order if necessary. The optimized version also controls the inner loop using a counter instead of a pointer, in order to avoid overhead through address calculations. Further improvements include early register overwrites (e.g. lines 17, 19 and 20) as well as loop unrolling (e.g. lines 9, 10, 16 and 25). The resulting code of the first optimization step is listed in figure 4. This code takes 2876 cycles to complete, which is a speedup of $\frac{5455}{2876} \approx 1,89$ compared to the original version.

This first version of the bubble sort algorithm can be optimized even further by introducing another if-else statement in oder to more rigorously minimize the amount of instruction inside the inner loop. Since the inner loop is executed quadratically many times, this is where the most code optimization has to take place, even if it means that more instructions have to be added outside the loop. Our final optimization, which is given in figure 5, only requires 2387 cycles to complete. This is a speedup of $\frac{5455}{2387} \approx 2,28$.

Listing 4: First version of optimized bs_begeu.s.

```
 1      ; Start BubbleSort (2876 cycles)
 2
 3      subu    %r4, %r3, %r2                   ; i = endIndex - startIndex;
 4      bgeu    %r2, %r3, outer_loop_end        ; if(startIndex <= endIndex)
 5      addui   %r1, %r0, $2                    ;       return;
 6
 7  outer_loop_begin:                           ; while(i != 0) {
 8      addu    %r7, %r0, %r10                  ;   ptr = array;
 9      lw      %r8, 0(%r10)                    ;       value = *ptr;
10      lw      %r9, 4(%r10)                    ;       value_next = *(ptr+1);
11      beqz    %r4, outer_loop_end
12      addu    %r5, %r0, %r4                   ;   j = i;
13      subui   %r4, %r4, $1                    ;   i--;
14
15  inner_loop_begin:                           ;   do {
16      lw      %r9, 8(%r7)                     ;       value_next = *(ptr+1);
17      bgeu    %r9, %r8, endif       ;r9 OLD   ;       if(value > value_next) {
18      addui   %r7, %r7, $4                    ;       ptr++;
19      sw      0(%r7), %r9           ;r9,r7 OLD          *ptr = value_next;
```

```
    sw      4(%r7), %r8         ;r7 OLD  ;           *(ptr+1) = value;
endif:                                   ;       }

    nop
    nop
    lw      %r8, 0(%r7)                  ;       value = *ptr;
    bgeu    %r5, %r1, inner_loop_begin   ;   } while(j >= 2);
    subui   %r5, %r5, $1                 ;       j--;

    j       outer_loop_begin             ; }
    nop

outer_loop_end:
```

Listing 5: Second version of optimized bs_begeu.s.

```
    ; Start BubbleSort (2387 cycles)

    subu    %r4, %r3, %r2                ; i = endIndex - startIndex;
    bgeu    %r2, %r3, outer_loop_end     ; if(startIndex <= endIndex)
    addui   %r1, %r0, $2                 ;       return;
    addu    %r7, %r0, %r10               ; ptr = array;

outer_loop_begin:                        ; while(i != 0) {
    lw      %r8, 0(%r10)                 ;   value = *ptr;
    lw      %r9, 4(%r10)                 ;   value_next = *(ptr+1);
    beqz    %r4, outer_loop_end
    addu    %r5, %r0, %r4                ;   j = i;
    subui   %r4, %r4, $1                 ;   i--;

inner_loop_begin:                        ;   do {
    bgeu    %r8, %r9, swap               ;     if(value < value_next) {
    lw      %r9, 8(%r7)                  ;       value_next = *(ptr+2);
    addu    %r8, %r0, %r9       ;r9 OLD  ;       value = value_next;
    j endif
swap:                                    ;     } else {
    addui   %r7, %r7, $4                 ;       ptr++;
    sw      0(%r7), %r9         ;r7,r9 OLD       *ptr = value_next;
    sw      4(%r7), %r8         ;r7 OLD  ;       *(ptr+1) = value;
endif:                                   ;     }

    bgeu    %r5, %r1, inner_loop_begin   ;   } while(j >= 2);
    subui   %r5, %r5, $1                 ;       j--;

    j       outer_loop_begin             ; }
    addu    %r7, %r0, %r10               ; ptr = array;

outer_loop_end:
```

a) *Attach your fastest-but-still-correct "bs_bgeu_opt????.s" to the mail.*

The final optimization is attached as *bs_bgeu_opt2387.s*.