Name Solution ____________________

Computer Architecture

EE 4720

Midterm Examination

Friday, 25 October 2002, 10:40–11:30 CDT

Problem 1	______	(30 pts)
Problem 2	______	(13 pts)
Problem 3	______	(13 pts)
Problem 4	______	(44 pts)
Exam Total	______	(100 pts)

Alias Who's in the Noose! ____________________

Good Luck!

Problem 1: A new MIPS branch instruction, `bieq rt,(rs) disp` (branch indirect equal) compares a register value to a memory location, if they are equal the branch is taken. The target is computed in the same way as other branches. In the code below, the contents of register `$s1` is compared to the contents of the memory location at address `$s2`. Like all MIPS control transfers, `bieq` has one delay slot. [30 pts]

(*a*) Modify the pipeline so that it can execute this new instruction. Show comparison units, multiplexors, and wires. **Do not** show control logic. For **partial credt** replace `bieq ...` with `bneq $s1,$s2, LOOP`.

See next page for solution discussion.

☑ Use as much existing hardware as possible. **Do not** add a new memory port.

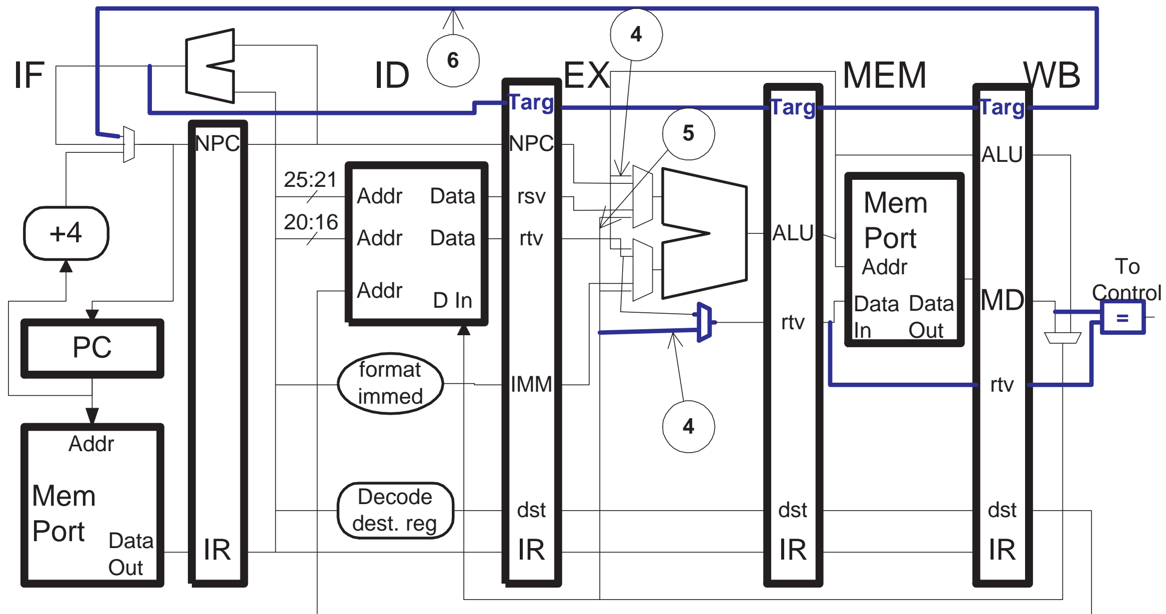
☑ The change should not reduce the clock frequency.

☑ Include the comparison unit for the branch condition.

☑ Add bypass paths so that the code below executes as shown.

☑ Label bypass paths with the cycle in which they are used. Include existing and any added bypass paths.

☑ Label the path carrying the branch target address with the cycle in which it is used.




```
LOOP:  # Assume bieq always taken.   # Solution
 # Cycle                0    1    2    3    4    5    6    7    8    9    10
 srl  $s1, $s1, 1       IF   ID   EX   ME   WB             IF
 addi $s2, $s2, 1            IF   ID   EX   ME   WB
 bieq $s1, ($s2), LOOP            IF   ID   EX   ME   WB
 sw   $s1, 100($s2)                    IF   ID   EX   ME   WB
 add  $0, $0, $0                            IF   ID   EXx
 sub  $0, $0, $0                                 IF   IDx
 or   $0, $0, $0                                      IFx
 xor  $0, $0, $0
```

Problem 1, continued:

(*b*) The pipeline execution diagram on the previous page only shows the first iteration.

☑ Continue the pipeline execution diagram to the beginning of the second iteration (when `srl` is fetched), *consistent with your solution to the first part.*

See diagram on previous page.

☑ Show which instruction is in each stage of the pipeline a cycle eight. Include squashed instructions, if any.

```
Solution:
 IF: addi    ID: srl   EX: or(sq)   MEM: sub(sq)   WB: add (sq)
 sq indicates the squashed remains of an instruction.
```

☑ Determine the CPI for a large number of iterations.

Iterations start at cycles 0 and 7. There are no stalls or other reasons why later iterations would be any different, therefore the number of cycles per iteration is 7. The loop has 4 instructions, for a CPI of $\frac{7}{4}$.

Part (a) discussion.

Changes shown in **blue bold**. The branch condition is resolved in WB when one of the data items is retrieved from memory. (A MEM-stage comparison would stretch the critical path, lowering clock frequency.) In addition to the memory value, the value of the `rt` register is needed, a new path from MEM to WB is added for that, as well as an EX-stage bypass path. The existing ID-stage adder is used to compute the target address, which is sent through the pipeline and used in the WB stage. (An alternative design would use the ALU to compute the branch target, but that would require a mux at the memory address input to select the address, which might be held in a new `rsv` pipeline latch.)

The circles show the cycle numbers. Note that the arrows point unambiguously to the path that's being used. (In some solutions the arrows would point to a bypass path, say, coming from memory before it reached the upper ALU mux, so one could not tell if it was a bypass into the upper or lower ALU mux.

Note: the problem as originally given did not explicitly mention clock frequency and the diagram had a path from MEM to the IF-stage mux.

Problem 2: Convert the MIPS program below to SPARC, making use of condition codes. [13 pts]

- [x] Make reasonable guesses about instruction names. Reasonable guesses will receive full credit.
- [x] For the branch instructions, explain what the name stands for.
- [x] Explain how each line of code uses the condition codes.
- [x] Use the minimum number of registers.
- [x] Do not rearrange instructions.

```
sub $s6, $s1, $s2
blt $s6, TARGET1
and $s3, $s6, $s4
beq $s3, $0, TARGET2
nop
beq $s6, $0 TARGET3
add $s5, $s6, $s3
```

Discussion: In SPARC, condition-code versions of arithmetic and logical instructions (such as `addcc`, `orcc`) set the condition codes based on the operation result, this is in addition to writing the result in the specified register. So the first instruction in the solution writes the difference in register `l6` and also writes the condition code register. For this first instruction the result of the subtraction is needed, by the `add` instruction.

The condition code register consists of four bits, Zero, Negative, Carry, and Overflow. Each is set based on the result of the operation.

```
subcc %l1, %l2, %l6  ! l6 = l1 - l2
bl TARGET1           ! Branch if < 0. (Negative, mnemonic assumes subtract.)
andcc %l6, %l4, %l3  ! l3 = l6 & l4
be TARGET2           ! Branch if equal to zero.
nop
subcc %l1, %l2, %g0  ! Set condition codes again. MIPS doesn't have to.
be TARGET3           ! Branch if equal to zero.
add %l6, %l3, %l5    ! Ordinary add.
```

Problem 3: Sometimes it's necessary to jump to a location specified in the program. [13 pts]

(*a*) SPARC V8 includes an instruction that can jump anywhere in the address space using an address specified in the instruction, for example, `call 0xabcd1234`.

☑ How does the `call` instruction, which is 32 bits, manage to specify a 32-bit jump target?

Because instruction addresses are aligned the least significant two bits of a jump target are always zero, so the instruction does not need to store them. The opcode in a SPARC instruction is just two bits, so the instruction can store a 30-bit immediate. (That immediate is a displacement, not the high 30-bits of the target address, but it can still jump anywhere in a 32-bit address space.)

☑ Switching now to MIPS, show the minimum number of instructions needed to jump to address `0xabcd1234`. (If `jr` is used the instructions must put the address in a register. The address cannot be loaded from memory.)

The solution appears below, followed by common mistakes.

```
# Correct
lui $l0, 0xabcd
ori $l0, $l0, 0x1234
jr $l0

## Common Mistakes Below

# Partial credit, could have used lui to reduce number of instructions.
addi $l0, $0, 0xabcd
sll  $l0, $l0, 16
ori $l0, $l0, 0x1234
jr $l0

# Partial credit, could have used lui to reduce number of instructions.
# Wrong shift amount
addi $l0, $0, 0xabcd
sll  $l0, $l0, 4      # WRONG, a hexadecimal digit is four bits.
ori $l0, $l0, 0x1234
jr $l0

## Common Mistakes Above
```

(*b*) Without introducing a new format, add a new instruction to MIPS that would allow a jump to an arbitrary address using two instructions. The address must be specified in the instructions themselves.

The new instruction is `jri rs, IMMED16` (jump register immediate). It jumps to an address constructed by oring the contents of register `rs` with `IMMED16`. It is a Type I instruction.

☑ Use the instruction in an example.

```
lui $s0, 0x1234
jri $s0, 0x5678  # Jumps to address 0x12345678.
```

☑ Show the coding for the new instruction.

Type I with low 16 bits of target address in the immediate field and the register number in the `rs` field.

Problem 4: Answer each question below.

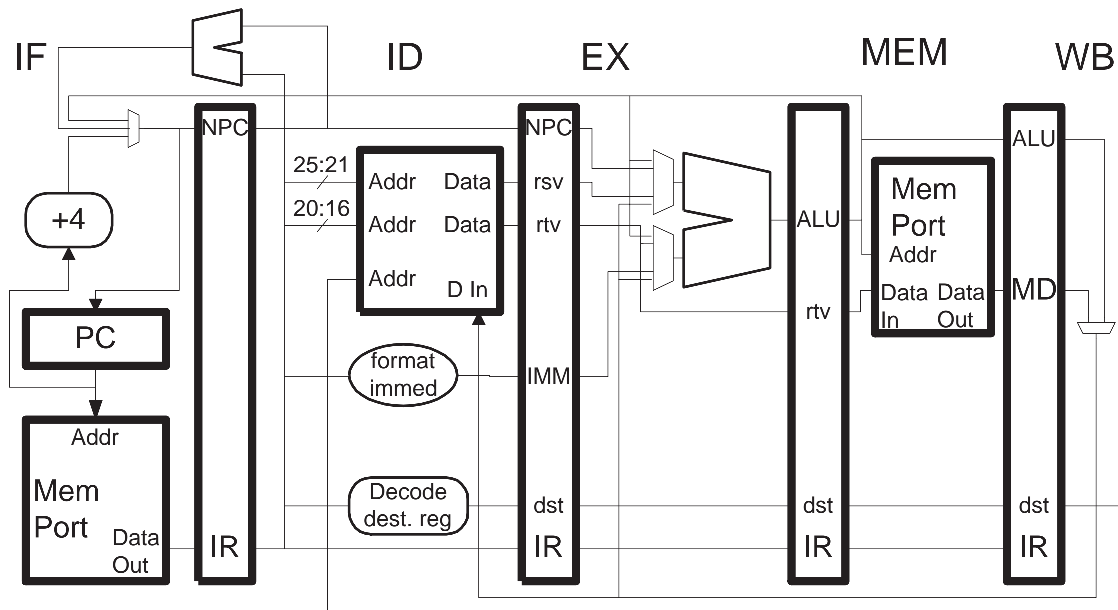
(*a*) Certain RISC ISA features are intended to facilitate pipelined implementations. [12 pts]

☑ Using the diagram below, briefly explain how RISC implementations benefit from fixed-length instructions and how things would be different with variable-length instructions.

Fixed length instructions allow implementations to easily compute the fetch address and to easily decode the instructions once fetched. With variable-length instructions the decode hardware might have to consider the instruction type before fetching registers. For example, different size formats might have the register numbers in different places, and so multiplexors would be needed at the input to the register file address ports. The IF stage could not just add 4 to get the next fetch address, the amount to add would have to be supplied by ID (if not slowing the fetch rate, requiring a shift-and-mask circuit to properly position the instruction).

☑ Using the diagram below, briefly explain how RISC implementations benefit from load/store instructions (restricting memory access to load instructions) and how relaxing the restriction would complicate things.

If source operands for ALU instructions could come from memory then either another memory stage would have to be added (adding to cost) or instructions would have to use the MEM stage before the EX stage, adding expensive new paths and greatly complicating the control logic (think about the logic for setting bypass paths).



(*b*) Before there was MMX there was BCD. [12 pts]

☑ Why are binary coded decimal (BCD) and packed-integer data types superficially similar? What is an important difference between them?

Discussion: A misconception (at least on the exam solutions) is that there are four integers packed into a register. The number of integers packed varies with the size of the integer (and the register). For example, a 64-bit register can hold 8 8-bit integers, 16 4-bit integers, etc. BCD digits are always four bits.

Answer: They are superficially similar because a decimal digit in a BCD representation might resemble a 4-bit number in a packed integer representation. An important difference is that the digits in a BCD representation are for one number whereas a packed integer holds a collection of numbers.

☑ Describe a computation in which packed integer data types yield a performance advantage over ordinary integers. Show a brief advantage of the computation.

```
// Part of solution.
int *a, *b, *c;  // Elements range from 0 to 15.
for(i=0; i<1024; i++) c[i] = a[i] + b[i];
```

The loop above compiled for an ordinary ISA would perform one addition per `add` instruction. If the compiler targeted a packed-operand ISA it might use an add instruction that adds two packed operands, each, say, holding sixteen numbers. (The compiler would have to know that numbers stored in the array are limited to 0-15.) The resulting code would require one sixteenth iterations. (Other code which accessed the arrays would also have to be written to handle the packed format.)

☑ Why were BCD data types added to some ISAs?

To avoid certain rounding errors.

(*c*) Describe two compiler optimizations that reduce instruction count. [8 pts]

Dead code elimination. If, say, the value of a variable is never used no instructions will be emitted for the code that computes a value for the variable.

Common subexpression elimination. If an expression appears multiple times then instructions for it might be emitted in one place, the instructions would store a value that would be used for each subsequent appearance of the expression.

(*d*) The BAPCO benchmarks used by PC magazine and others evaluates a computer by timing the execution of a suite of benchmarks. The suite of benchmarks is drawn from common applications, such as Adobe Photoshop. The applications are in executable form, source code is not used.

Like BAPCO, SPEC CPU2000 consists of a suite of common applications, unlike BAPCO source code is used. Assume that the BAPCO and SPEC benchmarks are both well chosen and aimed at roughly the same user community. [12 pts]

☑ How is source code used in preparing the SPEC CPU 2000 benchmark results?

The source code is carefully compiled for the system it will run on. Since manufacturers prepare the benchmarks for their own systems and knowledgeable customers respect the SPEC benchmarks every effort is made to select compiler options for the best possible compilation.

☑ Consider two processors, A and B. Suppose BAPCO ranks A faster but SPEC ranks B faster. How might the use of source code account for this difference?

Processor B had special instructions or scheduling characteristics that a compiler could take advantage of. Since SPEC benchmarks are compiled for the target system the results reflect their benefit. The BAPCo benchmarks are compiled by the software vendors and they target an average system, probably not taking advantage of B's special features. Without the special features A is faster.

☑ Taking in to account the differences due to the use of source code, who should make buying decisions based on BAPCO results? Who should make buying decisions on SPEC results?

Users buying "shrink wrapped" (or at least already compiled) software should use BAPCO results, especially if their applications are similar to BAPCO's. Those compiling their own applications or those buying code compiled for their particular system might use SPEC results.