Name David M. Koppelman

Computer Architecture
EE 4720
Final Examination

11 May 1998, 10:00–12:00 CDT

Modified

Problem 1		(20 pts)
Problem 2		(30 pts)
Problem 3		(20 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias The Solution

Good Luck!

Problem 1: An extended version of DLX is to include a new *morph* instruction, mnemonic `mrph`. Morph takes two arguments, an instruction address and a new instruction. After executing `mrph IADDR INSTR`, when execution reaches IADDR instead of executing the instruction at IADDR, INSTR is executed. This substitution only happens once, if execution reaches IADDR a second time the instruction at IADDR executes normally (unless `mrph` was executed again). For example consider:

```
 mrph POINTA, [subd f0, f2, f6]
 ...
LOOP:
 subi r2, r2, #3
POINTA:
 muld f0, f0, f2  ! On the first iteration subd will execute.
 bneq r2, LOOP
 addd f0, f0, f4
```

In the first iteration of the loop the `subd` instruction specified by `mrph` will be executed instead of the `muld`. After that the loop will execute normally. Morph might be useful for debuggers (the substituted instruction would be a jump to a debug routine).

A system can have at most one morph active at any time. The morph instruction cannot modify memory[1].

(*a*) Determine a format for the morph instruction that fits naturally into the DLX ISA. (An instruction fits naturally if it uses an existing type and its implementation requires little new hardware.) The format should include the type and how the arguments are specified, that is how mrph's arguments relate to IADDR and INSTR. In other words, how is the address specified (not too difficult since many existing DLX instructions specify addresses) and how is the instruction specified (the interesting part). (The three DLX formats types are type-R, used by `add r1,r2,r3`; type-I, used by `addi r1,r2,#3`; and type-J, used by `j LOOP`. Don't confuse the assembly language instruction with the actual instruction format, the assembly language form used above may be misleading.) *Hint: If you're not sure what to do in this part, attempt the next part first.* (5 pts)

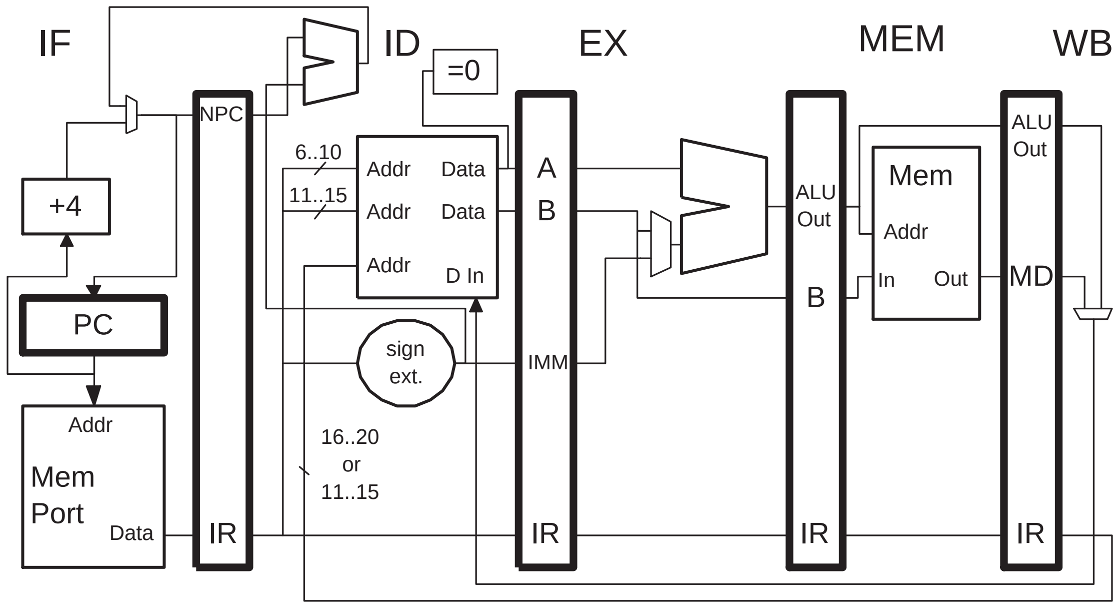
The morph instruction needs another instruction and an address. To fit "naturally" into the DLX ISA mrph would have to be 32 bits, so there would be no way to literally have the address and instruction present. Two options for the address are displacement addressing (as with branches) and register-indirect (as with some jumps). Register indirect is chosen because it does not require addition and because it leaves a source register operand free. The instruction (INSTR) is specified by a register, that is, the instruction itself is stored in a register. Since the type-R format has two source registers, it will be used.

In summary, mrph is a type-R format instruction in which register rs1 holds the address of the instruction to replaced (IADDR) and rs2 holds the instruction to substitute (INSTR).

[1] Morph could be implemented in software by replacing the instruction at IADDR by a jump to a morph routine. The morph routine would execute the substituted instruction and then return.

(*b*) Modify the pipeline below to implement **mrph** using the format chosen above. The pipeline must always execute the code below correctly. (The code itself is correct.) (For partial credit if your design will not execute the code below correctly explain why and how it might be fixed.) (15 pts)

```
 bneq r10,L1
 mrph A,[add r1,r2,r3]
 j L2
L1:
 mrph SKIP,[add r0,r0,r0]
L2:
 ...
 bneq r1, SKIP     ! Hint: May cause trouble.
A:
 sub r1, r2, r3    ! Maybe add r1, r2, r3 substituted.
 lw r4, 0(r1)      ! Hint: May cause trouble.
SKIP:
 addi r4, r4, #12  ! Maybe add r0, r0, r0 substituted.
```



Preliminary: The added hardware performs two actions: the execution of mrph, and substituting the instruction as specified by the last mrph. To execute mrph the presence of mrph is detected in the MEM or WB stage, if present IADDR and INSTR are clocked into special registers added for morph. (The values must be copied because the register contents can change.) A **morphactive** bit is also set. If the steps above were done in ID there would be no way to cancel the execution of mrph, for example if the preceding instruction encountered a page fault.

The hardware that performs the substitution is in the IF stage. The PC address which is being used for instruction fetch is also compared with the stored morph IADDR. If they match and morphing is active then the stored morph INSTR is clocked into the IF/ID.IR register rather than the value retrieved from memory and the **morphactive** bit is reset.

Solution continued on next page.

To implement morph the stage latches have an extra bit, called the M bit. This bit is set to 1 in IF if the substitute instruction is clocked into IF/ID.IR. The bit is passed down the pipeline unchanged. If a substituted instruction is cancelled and the M bit is 1 the `morphactive` bit is set back to one. Two examples of morphed instructions being cancelled appear in the code fragment above. Suppose the branch before `A` is taken after executing the first morph. While the branch is in ID the `sub` will be in IF and will be replaced with the `add`, the M bit will be set to 1, and the `morphactive` bit will be reset. Since the branch is taken the `add` (nee `sub`) will be cancelled in the next cycle. Since the M bit will be one, `morphactive` will be set back to one. Next consider the second morph above. Suppose the `lw` encounters a page fault. The morphed instruction `add` (nee `addi`) will be in EX when the fault occurs and (depending on implementation) in EX or MEM when cancelled. Again, because of the M bit the `morphactive` bit can be set back to 1.

In the original exam, the question asked "what might go wrong in the code above," in the modified exam that was changed to "if your design will not execute the code correctly explain why." Many had misinterpreted the original phrase to mean: "The programmer who wrote the code was careless and so the code has bugs. What did the programmer intend and what will go wrong?" There was nothing in the question to indicate that the program was other than correct, so the interpretation was not correct.

Problem 2: The program below executes on a 2-way, dynamically scheduled superscalar processor. The processor's features are summarized in the list below, the table gives details of the functional units. (The load/store unit computes the address in its first cycle and does the actual access in its second cycle. An operation does not enter the load/store unit until all its operands are ready.)

- ◇ Two-way superscalar instruction issue.
- ◇ Dynamically scheduled (see table), register renaming.
- ◇ CDB can accommodate results from any two functional units in any cycle.
- ◇ Reorder buffer for speculative execution and precise exceptions.
- ◇ Reorder buffer can retire as many as two instructions per cycle.
- ◇ Reservation stations, *not* reorder buffer, used for renaming.
- ◇ Zero-delay branch and branch target prediction. **No** branch folding.
- ◇ Branches do not have delay slots.

Name	Abbreviation	Latency	Initiation Interval	Reservation Station Nums
Load/Store	L	1	1	0-1
Integer	EX	0	1	2-3
F.P. Add	A	1	1	4-5
F.P. Mul.	M	5	2	6-7
Branch	BR	0	1	8-9
F.P. Divide	DIV	22	23	10-11

```
! When loop first entered r2-r1 large (loop iterates many times).
LOOP: ! LOOP = 0x1000
 lf   f0, 0(r1)     ! Don't overlook true dependencies on f0!
 mulf f0, f0, f2
 addf f0, f0, f3
 sf   4(r1), f0
 addi r1, r1, #8
 slt  r3, r1, r2
 bnez r3, LOOP
 div  f4, f5, f6
```

(*a*) Using the grid on the next page show the execution of the code above up to and including the last cycle shown on the grid. Assume perfect branch and branch target prediction, and no cache misses. Include instructions even if they have not yet finished executing at the end of the grid. (10 pts)

(*b*) Either determine and justify the CPI of an execution of a large number of iterations of the loop (ignoring cache misses and assuming perfect target prediction) or explain why it cannot easily be determined from the pipeline execution diagram. (A correct explanation of why it cannot be easily determined will get full credit, a correct CPI that left no time for problems 3 and 4 will get full credit for this subproblem and sympathy—but not credit—for omitting the others.) (4 pts)

In the execution shown on the next page (solution to first part) no two iterations are the same, thus the CPI is not easily determined given the severe time constraints imposed.

```
! First & Third Iteration
Cycle:           0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21

lf   f0, 0(r1)   IF  ID 0:L1 0:L2 0:WB
                                                                  IF   ID -------------> 1:LS 1:LS 1:WB
mulf f0, f0, f2  IF  ID 6:RS 6:RS 6:M1 6:M1 6:M2 6:M2 6:M3 6:M3 6:WB
                                                                  IF   ID -------------> 6:RS 6:RS 6:M1 6:M1 6:M2 6:M2 6:M3
addf f0, f0, f3      IF   ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS 4:A1 4:A2 4:WB
                                                                       IF  ------------->  ID 4:RS 4:RS 4:RS 4:RS 4:RS 4:RS
sf   4(r1), f0       IF   ID 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:RS 1:L1 1:L2 1:WB
                                                                       IF   -------------> ID --------> 1:RS 1:RS 1:RS 1:RS
addi r1, r1, #8           IF   ID 2:EX 2:WB
                                                                                           IF -------->   ID 2:EX 2:WB
slt  r3, r1, r2           IF   ID 3:RS 3:EX 3:WB
                                                                                           IF -------->   ID 3:RS 3:EX 3:WB
bnez r3, LOOP                  IF   ID 8:BR 8:WB
                                                                                                          IF   ID 8:BR 8:WB
dif fr,f5,f6                   IF   ID x
                                                                                                          IF   ID X
! Second & Fourth Iteration
Cycle:           0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21

lf   f0, 0(r1)                      IF   ID 0:LS 0:LS 0:WB
                                                                                                               IF   ID 0:L1
mulf f0, f0, f2                     IF   ID 7:RS 7:RS 7:M1 7:M1 7:M2 7:M2 7:M3 7:M3 7:WB
                                                                                                               IF   ID 5:RS
addf f0, f0, f3                          IF   ID 5:RS 5:RS 5:RS 5:RS 5:RS 5:RS 5:RS 5:A1 5:A2 5:WB
                                                                                                                    IF   ID
sf   4(r1), f0                           IF   ID --------> 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:RS 0:L1 0:L2 0:WB
                                                                                                                    IF   ID
addi r1, r1, #8                               IF -------->   ID 2:EX 2:WB
                                                                                                                         IF
slt  r3, r1, r2                               IF -------->   ID 3:RS 3:EX 3:WB
                                                                                                                         IF
bnez r3, LOOP                                                IF   ID 8:BR -->  8:WB

dif fr,f5,f6                                                 IF   ID x
Cycle:           0   1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20   21
```

(*c*) Suppose the second time the load (`lf`) executes it triggers a page fault exception, perhaps due to a bad load address.

In what cycle will the exception occur? Show the contents of the reorder buffer at that cycle and place a check mark next to instructions that have completed execution.

Explain how the reorder buffer will allow the exception to be precise. Specify what happens to the reorder buffer as a result of the exception, when it happens, and the contents of the buffer before and after it happens. At what cycle will the trap be inserted in the pipeline? (If solutions to the previous parts are not ready, make up a pipeline execution diagram just for this question.) (8 pts)

The exception occurs in cycle 7, when the second execution of `lf` is in L2 (MEM).

The reorder buffer at cycle 7 appears to the right, with oldest instructions at the bottom. The faulting instruction is marked with a ⋆.

addf
mulf
lf ⋆
bnez ✓
slt ✓
addi ✓
sf
addf
mult

The reorder buffer maintains precise exceptions by insuring that all instructions before the faulting instruction (`lf` in this case) complete while ensuring that no instructions following the faulting instruction can change anything. Permanent changes are made when instructions retire (write results and leave the reorder buffer). Retirement must proceed in program order and can only occur after the instructions complete. If an instruction at the bottom of the reorder buffer has faulted, all following instructions are deleted and a trap is inserted in the pipeline. It's important that the faulting instruction reach the bottom of the reorder buffer before inserting the trap so that the trap starts when all preceding instructions have committed. Based on the two-instruction-per-cycle retirement rate the `lf` reaches the bottom of the reorder buffer at cycle 16, the contents before all following instructions are deleted is shown to the right.

addf
mult
lf
bnez ✓
slt ✓
addi ✓
sf
addf ✓
mulf ✓
lf ⋆

(*d*) What are the minimum number of reservation stations of each type needed to attain a minimum CPI on the code above? What is that CPI? (There is a grid on the next page to work this out, other methods may be faster.)(8 pts)

The number of reservation stations is found by completing a pipeline execution diagram up to when every instruction in the first iteration finishes. The minimum number of reservation stations (without stalling execution) is used. The numbers are: L, 4; M, 3; A, 3; EX, 2; DIV 0, and BR 1. With at least these many reservation stations execution will proceed at $4/7$ CPI.

First Iteration, Third Iteration, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`lf f0,0(r1)`																					
`mulf f0,f0,f2`																					
`addf f0,f0,f3`																					
`sf 4(r1),f0`																					
`addi r1,r1,#8`																					
`slt r3,r1,r2`																					
`bnez r3,LOOP`																					
`div f4,f5,f6`																					

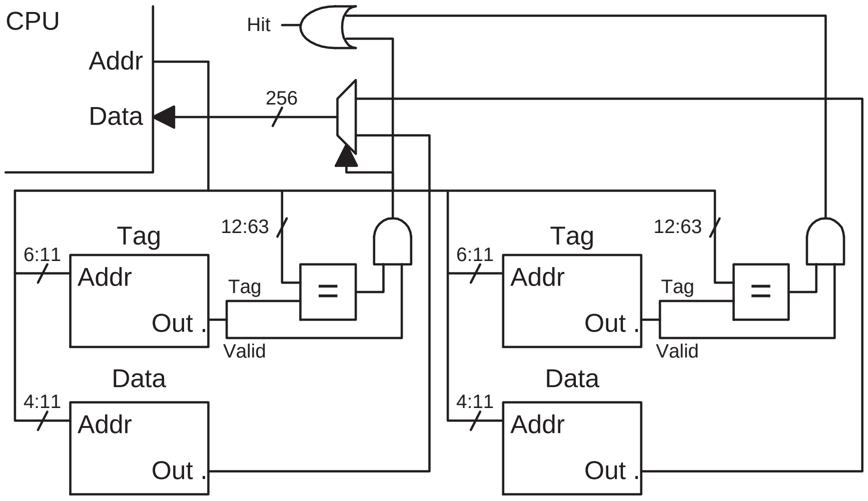
Second Iteration, Fourth Iteration?, etc.?

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`lf f0,0(r1)`																					
`mulf f0,f0,f2`																					
`addf f0,f0,f3`																					
`sf 4(r1),f0`																					
`addi r1,r1,#8`																					
`slt r3,r1,r2`																					
`bnez r3,LOOP`																					
`div f4,f5,f6`																					

Problem 3: A system has a 64-bit address space ($a = 64$), addresses 16-bit characters ($c = 16$), and has a 256-bit data bus ($w = 256$).

(a) Show how memory devices could be connected to construct a 2-way set-associative cache with 1024-bit lines and 64 sets. Memory devices of any size can be used, be sure to specify their sizes ($e.g.$, $x\,\mathrm{b} \times 2^y$). Show only the connections needed to retrieve the data and tag information, determine if the access is a hit, and pass the data to the CPU. (10 pts)

Each of the two data memories are 256×2^8 bits. Each of the two tag memories are 53×2^6 bits.



(b) Suppose the cache is write-through. What is the capacity of the cache and how much memory is needed to implement it? Be sure to specify units. (5 pts)

The capacity of the cache is $1024 \times 2 \times 2^6 = 2^{17} = 131072$ bits or $2^{14} = 16,384$ bytes or $2^{13} = 8192$ characters. To implement it $(52 + 1) \times 2 \times 2^6$ additional bits are needed. (No storage is needed for a dirty bit.)

(c) Find the addresses specified below. (5 pts)

- *Three* different address that are part of the same line and require exactly two loads to access. (Assume there are 256-bit load instructions.)

All have identical index and tag, two have same offsets but different alignment: For example: 0x0, 0x1, 0x10.

- Two addresses that can be in different lines but the same set.

Same indices, different tags. For example: 0x0, 0x1000.

- Three addresses that are in different sets.

Different indices. For example, 0, 0x40, 0x80.

Problem 4: Answer each question below.

(*a*) What is branch folding? In the implementations of DLX covered in class, why must the predictions used for branch folding always be correct? (6 pts)

Branch folding is the substitution of a branch target for a branch so that the branch is never seen further down the pipeline. It is implemented using a branch target buffer which holds the predicted target address *and* the target instruction. The prediction must be correct because in the DLX implementations a folded branch instruction does not get a chance to execute so there is no way to detect an incorrect prediction. Because predictions must always be correct branch folding is only appropriate for unconditional branches, which are usually called jumps. Systems with special branch units and flexible issue hardware can fold conditional branches, the branches execute in the branch unit leaving a bubble (wasted issue slot).

(*b*) Why might the cost of a functional unit with an initiation interval of 2 be less than one performing the same operations but with an initiation interval of 1 but having the same latency? Given such a cost relationship, what should the minimum number of reservation stations be for a functional unit with an initiation interval of ι and a latency of λ? Explain. (6 pts)

With an initiation interval of 2 only half the number of functional-unit segments may be needed, data would make two passes through each segment. We want to keep all the functional-unit segments busy (we could have gotten a cheaper functional unit with fewer segments, but we didn't so we must have had a good reason) so we would need at least $\lceil(\lambda+1)/\iota\rceil+1$ reservation stations. The $+1$ is for write back.

(*c*) What are some difficulties that might be encountered in developing a superscalar implementation of a stack ISA? (For partial credit, list some distinguishing features of a stack ISA.) (6 pts)

Stack ISAs typically have a variety of instruction sizes, making it difficult to fetch and decode several of them simultaneously. Code for stack ISAs have many consecutive instructions with true dependencies, forcing an implementation to rely on dynamic scheduling.

(*d*) Explain how a reservation register can be used to detect MEM-stage structural hazards while an instruction is in the ID stage. (6 pts)

A reservation register is a shift register indicating the status of a resource, in this case the MEM stage. Each position in the register is for a different number of cycles in the future. At each cycle the shift register is clocked, maintaining the time relationship. An instruction in ID checks the reservation register to see if the MEM stage will be free when it needs it. If so, it writes a 1 corresponding to the time it will be using MEM and will move to its functional unit in the next cycle. If not, it waits for the next cycle to try again (stalling the pipeline).

(*e*) What is the difference between a RAW hazard and a true dependency? (6 pts)

Pipelines have hazards, instructions have dependencies.