Boston University

College of Liberal Arts

Computer Science Department
111 Cummington Street, MCS 138
Boston, Massachusetts 02215
Tel: (617) 353-8919 / Fax: (617) 353-6457

CLA CS-450
Computer Architecture

Fall 1995 Homework 5
Due Thursday November 16, 1995

The Data Path shown in figure 1 is that of a simplified MIPS design. The processor uses 3 buses: S1, S2, and Dest. The fundamental operation of the datapath could be described in the following 5 execution cycles (stages):

Fetch Cycle
Fetch into the IR the instruction to which the PC is pointing in the code memory and increment the PC to point to the next instruction.

Decode and Register Read Cycle
Decode the instruction in the IR and load the A & B registers of the datapath with the contents of the registers identified by the *rs1* and *rs2* fields of the IR.

Execute Cycle
Data from two (or only one) of the registers (A, B, PC, IAR, TEMP, MAR, MDR) is operated upon in the ALU and the results written back to one of the registers (C, PC, IAR, TEMP, MAR, MDR).

Memory Access Cycle
Data is either downloaded into register C from memory or uploaded from the MDR into memory. The specific memory word addressed is specified by the MAR.

Write-back Cycle
The contents of the C register is written back to the register file.

The register file has 32 General Purpose Registers and has two output ports. This means that 2 of its registers can be fetched in one clock cycle into registers A and B. The register file is designed such that R0 has a value that is *always* 0. The control lines feeding the Register File allow any two of the 32 registers to be read and deposited in registers A and B in one clock cycle. In addition the register file is allowed to store the value of register C in any one of the 32 registers.

Each one of the registers in the datapath is supplied with enough control lines to enable its value to be deposited or withheld from each one of the buses it is connected to. Also, each register is supplied with a control line that allows it to "load" its input.

The ALU carries many different operations (you are free to pick your own, but make sure that at least the usual Arithmetic and Logical Operations are included!)

The CPU is connected to two separate memory systems: one for code (instructions) and one for data. You can assume that all memory operations complete in one clock cycle.

The machine is designed as a 32-bit machine. All data registers as well as all of the ALU datapaths are 32 bits long. The data and code memories of the machine are both 32-bit byte-addressable. This means that any memory load (store) brings in (out) 32-bit words, which could start at any byte address. Both memories can be as large as $2^{26} = 64$ MegaBytes. Thus, all address registers (i.e. PC and IAR) are 26 bits wide.

The ISA of this machine is that of a typical Load/Store architecture. There is only one addressing mode for this machine (typical of many RISC architectures), namely *displacement addressing*. In particular, the effective address of any memory access is computed by adding the contents of a register to an immediate value (provided in the instruction).

There are three types of instructions: R-type instructions, I-type instructions, and J-type instructions. The formats of these three types of instructions is shown below:

```
      6          5        5                     11
+----------+--------+--------+--------+-------------------+
| Opcode   |  rs1   |  rs2   |  rd    |        func       |
+----------+--------+--------+--------+-------------------+

                   R-type (Register-type)


      6          5        5                 16
+----------+--------+--------+----------------------------+
| Opcode   |  rs1   |  rd    |          Immediate         |
+----------+--------+--------+----------------------------+

                   I-type (Immediate-type)


      6                        26
+----------+----------------------------------------------+
| Opcode   |            Offset added to PC                |
+----------+----------------------------------------------+

                     J-type (Jump-type)
```

The R-type instructions are used to operate on the two source registers specified by the *rs1* and *rs2* fields, storing the result in a third register specified by the *rd* field. The *func* field of an R-type instruction specifies an extension to the *opcode* field. For example, a typical ALU operation (say ADD signed) will look like:

```
                        +--------+-------+-------+-------+-------------+
ADD R7, R5, R6   ==>    | 010010 | 00101 | 00110 | 00111 | 00001110110 |
                        +--------+-------+-------+-------+-------------+
                          opcode   rs1=R5  rs2=R6  rd=R7      func
```

The I-type instructions are used to operate on one source and on one immediate data item. They are typically used for load/store operations. For example,

```
                          +--------+-------+-------+------------------+
LW R16, R5(100)    ==>    | 111010 | 00101 | 10000 | 0000000001100100 |
                          +--------+-------+-------+------------------+
                            opcode   rs1=R5 rs2=R16   Immediate=100
```

The J-type instructions are primarily used for unconditional jumps to addresses relative to the PC, namely PC + 26-bit offset (specified in the instruction). For example, a typical Jump instruction will llook like:

```
                          +--------+----------------------------+
Jump 228           ==>    | 011011 | 00000000000000000011100100 |
                          +--------+----------------------------+
                            opcode            Offset=228
```

The instructions of the machine described above can be classified into the following categories:

Data Transfers
This includes instructions such as `LW` (load word) and `SW` (store word).

Arithmetic/Logical
This includes instructions such as `ADD, SUB, AND, OR, XOR` which operate on two register operands, storing the result in a third register. It also includes instructions such as `ADDI, SUBI, ANDI, ORI, XORI` which operate on one register and on one immediate operand, storing the result in another register. In addition, this category includes the "test-and-set" instructions such as `SEQ, SNE, SGT, SLT, SGE, SLE` which *set* or *reset* a register based on the result of comparing two other registers. For example `SEQ R5, R7, R9` will set `R5` to 1 if `R7` is equal to `R9`, otherwise `R5` will be reset to 0. There is an immediate version of the "test-and-set" instructions such as `SEQI, SNEI, SGTI, SLTI, SGEI, SLEI`, which compare a register to an immediate value and sets/resets another register based on the comparison.

Control
This category includes conditional branch instructions such as `BEQZ, BNEQZ` which branch to a location (computed as a 16-bit offset from the PC) if a particular register (specified in the instruction) satisfies a condition (e.g. Equal to zero, or not equal to zero). For example, the instruction `BEQZ R4, -248` means that if `R4` is equal to zero, then the next instruction to execute should be the one -248 bytes away from (248 bytes before) the current PC. Also, this category includes unconditional jump and jump and link instructions such as `J, JAL`. All jumps are to a 26-bit offset relative to the PC. Linking is done using `R31`.

Answer the following questions:

a. A fan of CISC architectures criticized the machine described above because it doesn't have a machine instruction for loading an immediate value into a General Purpose Register and because it doesn't have a machine instruction for clearing a General Purpose Register. As a fan of RISC architectures (by virtue of your instructor's bias ☺), how would you respond? In particular, give *one* instruction that could be used to load a register with an immediate value and *one* instruction that could be used to clear a register. What is the effect of not having these instructions on the readability of assembly language programs? Should you be concerned about it?

b. A fan of CISC architectures criticized the machine described above because it supports only one addressing mode (displacement addressing). Is that a big disadvantage? If yes then why? If not then why not? How would other addressing modes be supported? In particular, write the sequence of instructions needed to perform an indirect addressing load. What is the penalty (in terms of # of clock cycles, i.e. latency) for not supporting indirect addressing.

c. Write the RTL statements that would be needed to perform the 5 execution cycles for:

1. A Data Transfer Operation (do it for: `LW` and `SW`).
2. An ALU Operation (do it for:`ADD`,`XORI`, and `SGE`).
3. A Control Operation (do it for:`BEQZ`, `J`, and `JAL`).

d. Design a microprogrammed controller for this MIPS-like machine (notice that RISC machines have Hardwired controller units). In particular:

1. Identify all the control points in the datapath (preferably giving each one a symbol and describing its function).
2. Suggest a encoding mechanism for the control points (how many fields will be needed to control the datapath and how will these be encoded).
3. Suggest a microsequencing mechanism (Micro-PC control).
4. Suggest a microinstruction format.
5. Write the microprogram for the `ADD`, `LW`, and `BEQZ` instructions.

e. How easy/difficult would it be to pipeline your design? Discuss any changes you would need to make to the datapath and controller. (Do *not* redesign the machine, simply identify the problems and suggest possible venues to solve them). In view of the pipelined nature of the modified machine, what is the performance penalty for not supporting indirect addressing (in reference to part b. above).

Good Luck!

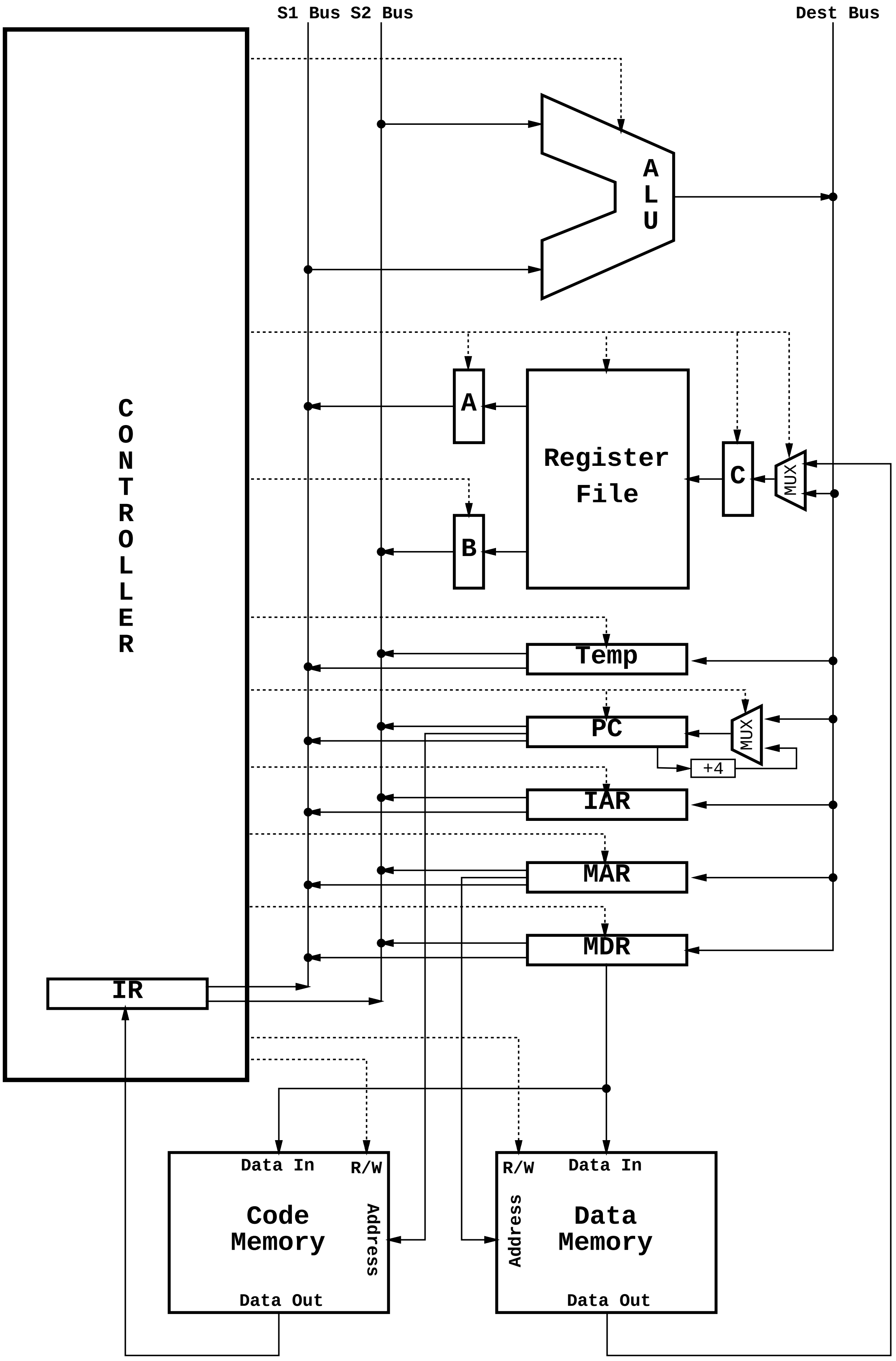

Figure 1: Simplified Data Path of a MIPS CPU