James Johnston
CS 6133
Module 5

Part 1

1 Give two examples of legacy stack based machines that were built and successfully used for commercial purpose. Provide architectural details that made them unique and practical.

The English Electric KDF9 machine was first delivered in 1964. The KDF9 had a 19-deep pushdown stack of arithmetic registers, and a 17-deep stack for subroutine return addresses. Its logic circuit were entirely solid state.

Tandem Computers T/16 was like the HP 3000. Compilers controlled the register stack when it spilled the memory stack or was refilled from the memory stack.

2 What type of internal storage does Dr. Neumann's digital computer use? Justify your answer with details.

Dr. Neumann's digital computer uses a read write random access memory to store program instructions and data. It is one of the first stored program computers to date. It also uses external mass storage and input and output control mechanisms.

3 Read section A.1 (5th ED.) or B.1 (4th ED.) in "Computer Architecture - A Quantitative Approach" and answer the following questions:

1 How are variables, used in a program, stored in a CPU?

Automatic variable have allocated storage in memory, but accessing data in memory is slower than processing in the CPU. These computer have small amounts of storage, called registers, within the CPU itself where data can be stored and accessed quickly.

The compiler determines the data that is to be stored in the registers of the CPU. The C language provides the storage class register so that a programmer can "suggest" a particular automatic variable that should be allocated to a CPU register. The register variables provide a certain control over efficiency of the

program. Variables that are used repeatedly or whose access times are critical may be declared as storage class register.

2 Is there an optimal number of registers a CPU should have? How would one calculate the optimal number of registers?

Studies have shown that the number of register has been limited to technology and bottle necks. Early designs were limited due to the number of transistors and circuit components. Today physical registers are limited by muxes and latency.

The number of registers available on a processor and the operations that can be performed using those registers has a significant impact on the efficiency of code generated by optimizing compilers.

3 What are the two types of CPUs with general purpose registers that are popular today? What type of internal storage do they use?

The 16 bit x86 is a CPU with general purpose register that is popular today. It uses PROM for internal storage.

The 32 bit x86 is a CPU with general purpose register that is popular today. It uses PROM for internal storage.

4 Do compilers play a role in a CPU's efficiency? if yes, how? Justify your answer with details

Yes, compilers play a role in a CPUs efficiency. Many good compilers know hoe to translate code to instructions efficiently. For example adding integers can be excused simply and efficiently using a good compiler. A well compiled application will tell the CPU to add the numbers and store them. But a poorly compiled program may include additional instructions that reduce the performance and slow the application down. Today many complex applications like graphic editors, video editors, etc. need to hundred of operations just to start. A good compiler must be able to be able to do several extension through shared functions. If it is done well it will require less CPU power and accomplish the best results.

Part 2

Consider the following C code:

```
1l int a[10], b[10], c[10], d=9, i;
2l
3l for(i=0; i<10; i++)
4l a[i] = (b[i] + c[i]) - (c[i] * d);
```

1 Write an equivalent assembly code for four different storage types described in the previous lecture (Section A.1 and A.2). For this part use only register and immediate addressing modes. You could invent your own assembly mnemonics (use figure A.2 for reference)

```
WIDTH=32; DEPTH=1024;
ADDRESS_RADIX=HEX; DATA_RADIX=BIN;
CONTENT BEGIN

000 : lw r0, 0      —load data memory 0 -> r0 for the i counter
001 : lw r1, 0          —load data memory 0 -> r1 for the index of array A
002 : lw r2, 0          —load data memory 0 -> r2 for the index of array B
003 : lw r3, 0          —load data memory 0 -> r3 for the index of array C
004 : lw r4, 9          —load data memory 9 -> r4 for d value
005 : lw r5, 10     —load data memory 10 -> r5 for the limit of the counter
006 : lw r6, 1      —load data memory 1 -> r6 for the increment
007 : lw r7, 0          —load data memory 0 -> r7 for the sum of b and c
008 : lw r8, 0      —load data memory 0 -> r8 for the product of c and d
```

```
009 : add r7, (r2), (r3) —add b[0] + c[0] and set it equal to r7
00A : add r8, (r3), r8       —add c[0] to r8 9 times to simulate multiplication
00B : add r8, (r3), r8       —add c[0] to r8 9 times to simulate multiplication
00C : add r8, (r3), r8       —add c[0] to r8 9 times to simulate multiplication
00D : add r8, (r3), r8       —add c[0] to r8 9 times to simulate multiplication
00E : add r8, (r3), r8       —add c[0] to r8 9 times to simulate multiplication
00F : add r8, (r3), r8            —add c[0] to r8 9 times to simulate
multiplication
010 : add r8, (r3), r8             —add c[0] to r8 9 times to simulate
multiplication

011 : add (r1), r7, r8       —(b[0] + c[0]) - (c[0] * d)


012 : beq r0, r5, 101—branch if equal to limit(10),go to END and break loop

013 : add r1, r6, r1         —add one to A index
014 : add r2, r6, r2         —add one to B index
015 : add r3, r6, r3         —add one to C index

016 : add r7, (r2), (r3) —add b[i] + c[i] and set it equal to r7
017 : add r8, (r3), r8       —add c[i] to r8 9 times to simulate multiplication
018 : add r8, (r3), r8       —add c[i] to r8 9 times to simulate multiplication
019 : add r8, (r3), r8       —add c[i] to r8 9 times to simulate multiplication
0AA : add r8, (r3), r8       —add c[i] to r8 9 times to simulate multiplication
0BB : add r8, (r3), r8       —add c[i] to r8 9 times to simulate multiplication
0CC : add r8, (r3), r8            —add c[i] to r8 9 times to simulate multiplication
0DD : add r8, (r3), r8             —add c[i] to r8 9 times to simulate
multiplication

0EE : add (r1), r7, r8       —(b[i] + c[i]) - (c[i] * d)
0FF : add r0, r6, r0         —add one to i
100 : beq r0, r0, 012—branch always equal ,go to add beginning
101 : out r1
 END;
```

2 Rewrite (assembly code) using an addressing mode that would yield the lowest number of instructions for each storage type.

```
lw r0, 10      —limit
```

```
lw r2, 9        —multiply
lw r3, 0        —i
lw r4, 0        —b[i] +c[i]
lw r5, 0        —c[i] * 9
lw r6, 0        —A

loop:  cmp r3, r0 —while r3<10

        je end

        add r4, b(r3), c(r3)

        muli r5, c(r3), 9

        add r6, r4, r5

        mov a(r3), r6

        inc r3

        jne loop

 END;
```

3 Let's say that a[0] is stored at memory address 0x00001000, b[0] at 0x00002003 and c[0] at 0x000040FE. Calculate the total number of memory read and write operations. Assume that memory is 64-bit wide, an integer is 32 bits and the architecture allows byte addressing.

Memory read:

a[0] — zero

b[0] to b[10] - ten operations

c[0] to c[10] - ten operations x 2

30 read memory operations

memory write:

a[0] to a[10] - ten operations

10 write memory operations

Part 3

Examine chapter 2, chapter 4 (page 67 - 73) and Appendix A in MIPS R4000's User's Manual2, read Appendix J3 and answer the following questions:

1 List and describe system and procedure call/return instruction supported by R4000 CPU.

The R4000 is a scalar superpipelined microprocessor with an 8-stage integer pipeline. In the first stage, a virtual address for instruction is created and the instruction translation look aside buffer translates the address to physical address. During the second stage, the translation is completed and the instruction is collected from an internal 8KB instruction cache. The instruction cache is direct-mapped, virtually indexed and physically tagged. It has wither a 16 or 32 byte line size and could be expanded to 32KB.

During the third stage, the register file is read and the instruction is decoded. The MIPS III defines 2 register files, the first for integer unit and the second for float-point. Register files are 64 bits wide and contain 32 entries. The integer register file has 2 read ports and 1 write port. The floating-point register has 2 read ports and 2 write ports. At stage 4 execution begins for both floating-point and integer instructions. Both are written back to the register files when completed.

2 How does MIPS CPU protect jump or branch into privileged/kernel memory?

In user-mode the program places values in registers, or creates arguments inside the stack frame to indicate what specific services it requires from the OS. The user-mode program then performs trap instructions. The CPU immediately switches to kernel mode and jumps to instructions at a fixed location in memory. These instructions have memory protections so that they cannot be modified by the user mode programs. and are unreachable.

The trap instructions read the details of the requested service argument and then preform the request in kernel mode. After the system call is done, the OS resets the mode to user-mode and returns from the system call. The CPU jumps in kernel mode to the system call handles which does the work and returns the program in user-mode.

3 Assume that R4000 does not have a floating point co-processor. Could a developer write an application that requires floating point data types? If yes, which instruction could she use and how?

As far as my knowledge of the system, I would have to say no. The R4000 has an on-die IEEE 745-1985-compliant floating-point unit. The FPU is a co-processor designated CP1. It creates two modes, 32 and 64 bit which are selected by the FR bit in the CPU status register.

In 32 bit mode the 32 floating-point register becomes 32 bits wide whatever I when used to hold single precision floating -point numbers. When used to hold double-precision numbers there are 16 floating-point registers.

The FPU can also operate in parallel with the ALU unless there is a data or resource dependency, which causes it to stall. There are three subunits: adder, multiplier and divider. The divider and multiplier can excite an instruction in parallel with the adder, but use the adder in the final stages of execution. Thus imposing limits to overlapping. Under certain conditions it can execute up to three instructions at a time, one in each. The FPU can retire on instruction per cycle. The adder and multiplier are pipelined and it is clocked at twice the frequency of the microprocessor. Division has 23-36 cycle latency for single and double precision ops.