CSE 410 - Computer Systems
Autumn 2001

Review sheet for midterm exam

The information on this sheet is intended to help you identify the key points that you should be comfortable knowing. It is not a substitute for the last month of lectures and homework assignments, and so it is possible that there will be questions on the exam that are not completely answered by the material on this review sheet.

The mid-term will be given in class, Wednesday, November 7.

Computer Instructions

Registers. Register names and usage convention for each register (see the register table in lecture Oct 8, p19). Which registers should not be used at all (at, k0, k1), which registers can be changed without saving and restoring them (a0-a3, t0-t9, v0-v1), which register is a constant ($zero), and which registers must be saved and restored if a procedure wants to use them (s0-s7, gp, sp, fp, ra). Instructions: move.

Load and Store Instructions. Memory address, what it means that memory is byte-addressable, byte, half-word, word, double word, and alignment in memory. Load and store word, load and store byte, difference between signed and unsigned loads, difference between aligned and unaligned loads/stores. Basic addressing mode: offset + base register value. Big-endian, little-endian storage convention. Immediate mode values (constants embedded in the instructions) and the limitations on their magnitude (limited by the size of the 16-bit field). Instructions: la, li, lb, lbu, lw, ulw, sb, sw, usw.

Branch instructions. Be able to read and interpret the various branch instructions. Limitations on the distance that a branch can go. PC-relative branch addressing. I will not ask about the comparison instructions (slt, etc) because I expect you to understand the branch instructions in the following list, and they do the comparison for you if needed. Instructions: b, beq, bne, bgt, bge, blt, ble, beqz, bnez, bgtz, bgez, bltz, blez.

Jump instructions. Understand that the jump and link instruction is the key to calling procedures, that the jump register instruction is the means for getting back, and how they interact through the $ra register. Plain jumps can go further than any branch due to the larger offset that they contain. Pseudo-direct addressing. Instructions: j, jal, jalr, jr.

Arithmetic and Logical instructions. Recognize that a trailing "i" indicates an "immediate value (a constant)", a trailing "u" indicates "unsigned (no overflow exception generated)". Be able to read a table of existing register values and an instruction to operate on some of those registers, and write down the result of the instruction.

Understand what happens when you do a "shift logical" to the bits in a register (bits that are shifted out are dropped, bits that are shifted in are zero, the rest of the bits are in a new position). Understand AND and OR operations, and what it means to use a mask. I wrote a long email about this to the list, see the archive. Instructions: add, sub, div, mul, and, or, sll, sllv, srl, srlv.

Writing assembly code. The .data directive tells the assembler to put data in the heap, .text tells it to put instructions in the program code section. Directives .word, .byte, .asciiz store data in memory. A label in the code or data sections is a symbolic reference to a location in memory. Know how to write comments that add meaning for the next programmer and are not just a translation of the instructions into words. Assembly code is the set of instructions that you write, machine code is the binary values that those instructions get translated into.

Procedures

Understand how program memory is laid out (program, heap, stack) and how the heap and the stack grow towards each other. Identify the steps in calling another procedure (set up parameters, transfer control, acquire needed storage, do the task, make result available, release storage, return) and understand how each step is accomplished. Registers for parameters, stack space for parameters, adjusting stack pointer to control stack usage, stack space for saving and restoring register values that must be preserved, $v0 to return a value. Given a small segment at the start of a procedure, draw the contents of the stack frame.

Leaf procedure, non-leaf procedure, calling tree diagram. Given a short sequence of code, draw the calling tree and label the arguments of the called procedures.

Numbers and Formats

Character data. It is common to store characters using 8-bit values that fit in a byte. Strings of such characters can be terminated by a null-byte (a zero value) or they can be counted with a separate variable holding the number of characters in the string. Other encoding formats such as Unicode use more bits per character and thus can encode larger numbers of characters.

Binary, hex, and decimal number bases. Total number of values in a field that is n bits wide is 2^n. Maximum value is 2^n-1. Starting from a binary, hex, or decimal value less than 16_{10}, convert it to the equivalent value in binary (0 to 1111_2), to hex (0 to F_{16}) or to decimal (0 to 15_{10}). Convert any length binary number to the equivalent hex value, and hex to binary. Know that the following value is the largest that will fit in eight bits:

$1111\ 1111_2 = FF_{16} = 255_{10}$.

Understand that shifting a number one bit position to the right divides by 2, shifting one position to the left multiplies by 2. See the Number Base Charts in the addendum for

October 5, for information about why the values are related the way they are. Hex numbers are often written with a leading "0x" to indicate that they are base 16.

Signed Integer Numbers. Understand 2's complement notation. The sign bit is the high order bit (most significant bit, bit 31 in a 32-bit quantity), it is 0 for positive numbers, 1 for negative numbers. Convert from negative to positive representation or vice versa using "complement and add 1." Construct a table like the one shown in the lecture of October 15, page 6, that shows the binary, hex, unsigned decimal, and decimal values for a set of 2's complement numbers. The field width will be 4 bits or smaller, so there won't be hundreds of numbers to write down.

Floating Point. Given a drawing of two 32-bit words, and the information that the field width of the sign bit is 1, the field width of the exponent is 11, and the field width of the mantissa is 52, show how the fields are placed in the word. Given a number written in "binary scientific notation" ($\pm$ 1.fraction x 2^{exponent}), be able to set the sign bit correctly, and show where the signed value is stored in the words (the leading 1 is not stored because it is always there, the fractional part is stored in the mantissa) and where the exponent is stored (in the exponent field).

Pipelining

The five stages of the basic MIPS pipeline are IF, ID, EX, MEM, WB. Instruction Fetch, Instruction Decode, Execute, Memory, Write Back. The instruction set was designed to be pipelined: The instructions are all 4 bytes, that simplifies IF. Instructions have common format (op code location, register locations), that simplifies ID. Only load and store instructions access memory, that simplifies MEM. Most memory operations are aligned, that simplifies MEM. Branch instructions are a problem for pipelines because they change the order of instruction fetches, which may have already started by the time the branch decision has been made. A strategy for coping with this is the "branch delay slot", in which an instruction is always executed after the branch but before the instruction at the new destination address. Another strategy is "static branch prediction" in which the programmer or the compiler makes a guess about which will happen more frequently. "Dynamic branch prediction" keeps records with a simple state table and builds a prediction based on program behavior. The cost of a mistake is the need to flush the instructions that are already in the pipeline, which slows it down.

Exceptions

An exception is an event that causes a change in the execution sequence of the program instructions, without using an explicit branch or jump instruction. An exception may be caused by an error, but exceptions are also used extensively in the normal operation of the computer. An exception causes the CPU to save a small amount of information about its current state, then transfer control to a location in the operating system that can handle the situation. Common causes of exceptions include external events (device interrupts), memory translation exceptions (needed memory page not loaded), system calls, and program errors (divide by zero). The system uses $k0 and $k1 to create a little working room for itself when an exception occurs, that is why the values may change at any moment and your program cannot use these registers.

Caches

All caches are based on the observation that code exhibits temporal and spatial locality, ie that which is in use now may be needed again soon, and that which is in use now probably has neighbors which will be needed soon. The cache is smaller than the storage that is behind it, and so it can be more expensive and hence probably faster. This strategy applies to main memory and also to any other storage system whose usage patterns exhibit temporal and spatial locality of reference, for example the web. Cache hit and cache miss refer to whether or not a particular item is in the cache when it is needed. The hit rate, miss rate, hit time and miss time, are critical values when sizing the cache and evaluating cache line replacement strategies. In high speed caches close to the CPU it is most important that the access be extremely fast, so the cache line replacement strategy tends to be fairly simple, often just a random selection of the item to replace. Further from the CPU (for example, the main memory acting as a cache for items on disk) more elegant strategies are appropriate.

A cache contains a set of key / value pairs.

In a direct mapped cache, the cache line index is directly calculated from the address (the key) of the memory location whose data (the value) is to be stored in the cache. Several different addresses map to the same cache line index, and so it is quite possible to have to evict a cache entry to make room for a new entry even though the cache is not full.

In a fully associative cache, the entire address (the key) is stored along with the data (the value) and when an entry is needed a parallel search is performed for the matching address. If the address is in the cache, the appropriate cache line responds and provides the data. A fully associative cache will always be filled (after initial startup) and will only evict an entry when space for a new entry is required.

An n-way set associative cache is a mix of the two styles listed above. The address (the key) is used to calculate the cache line index, but then a parallel search is performed on the small set of entries that are kept at that index. Slower than a direct mapped cache, but higher hit rate. Lower hit rate than a fully associative cache, but faster.

Spatial locality is addressed by bringing in blocks of data instead of just the one word that is needed for a particular reference. 64 words is typical block size for memory cache.

Write-through cache updates the backing storage every time a write occurs to the cache. This is effective at maintaining cache coherency, but can be very slow, because it runs at memory speeds, not cache speeds, when writing. Can be improved with buffering, if writing is sporadic and bursty.

Write-back cache only updates main memory when an item is being evicted from the cache and has been changed. Maintaining coherency with main memory can be complex, and is critical to the success of the design.

Review the chart "Summary: Classifying Caches" from the lecture of October 24, page 34. You should understand the questions and the choices.

Virtual Memory

Virtual memory is the key element that allows modern computers to have large address spaces and utilize them effectively. The program addresses that your program generates for instructions and data are considered to be virtual addresses. A certain number of bits in an address are dedicated to indicating the offset within a memory page. Typically this is 12 bits, which provides 4096 offsets (from 0 to 4095_{10}). As a result, virtual pages (and the corresponding physical pages) are typically 4KB long.

The operating system use the memory management unit to translate between program addresses (virtual addresses) and physical addresses. This is done by chopping off the low order 12 bits and setting them aside as the offset. In a machine with 32-bit addresses, the remaining 20 bits are considered to be the virtual page number. The virtual page number is used as an index into a giant array called the page table. Each page table entry contains the Physical Page Number that the system has assigned this virtual page to. The physical page number is concatenated on the front of the offset, and the result is the actual physical address where the value of interest is located.

Any virtual page (a page of program addresses) can be mapped to any physical page, or to none. This allows many benefits, including easily sharing main memory between multiple processes, memory protection, swapping pages to disk, effective handling of sparse address spaces, and sharing memory between processes.

If the physical page numbers all require fewer than 20 bits to store, then the amount of physical memory is less than the largest virtual address. If the physical page numbers require more than 20 bits to store, then they are bigger than the virtual page numbers and so the physical memory is bigger than the largest virtual address. The running program cannot tell the difference between these two conditions and is completely unaware of the size of main memory, although there may be speed issues in a small memory system.

Page tables are large, but since they are addressed using virtual addresses in the kernel, only the amount of memory needed is actually used. Also, the page table entries are cached in the translation lookaside buffer, and so they are usually very fast to reference.

Input / Output

Data is moved from various external devices to the CPU and main memory using the input / output devices. Devices are input, output, or both, and may have human partners (user interface) or machine partners (storage, data access).

Disk drives comprise one or more platters with a magnetic coating. The individual bit positions on the disk are read and written using a read/write head. The head is mounted on an access arm which moves over the surface of the disk, while the disk spins. The spin rate is typically 3600 RPM to 10000 RPM. The time it takes to get a block of data from the disk includes the following. The seek time is the time it takes the access arm to position the head over the correct track on the disk. This is generally on the order of 12 milliseconds average. The rotational delay is the time it takes the platter to rotate so that the data is under the read head. On average, this is one half of a revolution. This can be calculated as 60 * 1/RPM * 0.5, and is typically in the range of 3 to 8 milliseconds. The transfer time is the amount of time it takes the disk to move all the data on the disk under the read head. The actual data access times are improved because modern disk drives contain large amounts of memory dedicated to caching and buffering reads and writes.

Devices are often controlled using memory mapped registers. The memory translation system generates signals which are interpreted by the devices. If a device recognizes its address range, it responds by reading or writing data as though it were memory. Small amounts of data are transferred to and from devices using Programmed I/O, in which the processor reads and writes the data values as they become available. Large bursts of data can be transferred directly from devices to memory using Direct Memory Access (DMA) in which data is transferred from the device to memory without intervention by the CPU except perhaps at the beginning or end of the transfer.

Devices communicate with each other and the system using buses. The two primary constraints are speed and flexibility with a constant tradeoff between them. The memory to processor link must be fast, so it tends to be less flexible and more dedicated to a specific task. This bus is often synchronous, meaning that it runs at a constant clock rate and uses a simple protocol.

The device to memory link is much less well specified, and must be flexible to accommodate a wide variety of devices, both existing and future. Hence the bus is often asynchronous, meaning that there is not single clock signal and devices must use handshaking with several control signals to complete every transaction.

Arbitration is the term describing the decision about who will be allowed to be bus master (who controls the events in a transaction). Multiple masters are important for sharing the load evenly among devices, but arbitration adds complexity to the bus design.

Input/Output buses tend to become very entrenched once there are a lot of devices available for them, and consequently it is not unusual to see systems that have a variety of I/O buses included in them to accommodate several generations of external devices.