Kevin Martin
Syracuse University
CIS655 – Summer 2020, Tuesday @ 9:00pm EST
Final Exam – Take-home portion

0. Background
   - 0.1. I will be largely using the instruction set I proposed from Assignment 2. I believe that assignment went very well and like how the instructions were decoded and ran. At the time, I even began to conceptualize how the memory would be structured, at least in regard to how the registers would interact with it. The downside of the architecture is that it is not very efficient in a real-world setting. Each OpCode is too long, and there are more concise ways to express the various commands. However, I still believe it is a good framework to discuss the overall implementation goals, and indeed look at where a different decision may have been more beneficial.
   - 0.2. The architecture would be most appropriate for a low-level application, such as an embedded system. The system actually could support quite a few commands, 124 specifically, but was not designed for complicated tasks. The emphasis is on using all 16 registers as much as possible for quick speed and reducing the amount of memory trips as much as possible. The implementation could be further reduced with even less hardware, but as it stands the architecture would be better suited for something with low to moderate complexity. I would classify this system as something suitable for a basic calculator, modern microwave, or similar, which requires some logic applications and customization, but not an extensive amount. A basic security system may also be appropriate, but only one with sensors and a light notification system (as opposed to a more modern approach like Google's Nest). To help with simplicity, this system will only be available in 32-bit applications. No desktop/personal computer applications would ever need to be extended to this type of system.
   - 0.3. The system will be referred to as REOM: **R**ISC-based **E**xtended **O**pcode **M**icroprocessor

1. Identify a set of processor parameters and structures
   - 1.1. Processor components and characteristics for each component
     - 1.1.1. REOM would utilize a single coprocessor setup where the main coprocessor would be responsible for the majority of the system control and the second coprocessor would handle the floating-point arithmetic. This is similar to the MIPS implementation but without the ability to extend for application-specific instances. This keeps in mind the goal of lower cost, less size, and speed. The system would only require one single ALU to process arithmetic operations as there will not be enough complexity to justify any more dedicated hardware for this area (other areas will require specific hardware applications).
     - 1.1.2. REOM will use 16 registers which support 19-bit words. 19 bits is the minimum amount to cover every scenario: 2 bits to identify one of the four Types, 5 bits for the OpCode (32 operations for each of the 4 instruction types leads to an available 124 operations), and three places to access one of the 4-bit registers.
     - 1.1.3. As previously stated, the theoretical maximum number of instructions that could be supported is 124. However, based on the stated application, less than half that would be required. The decision to create 4 Types of instructions (identified by the first two bits of each word) was to categorize each of the four types: 11 for arithmetic, 01 for branch/control, 10 for memory access, and 00 for system calls. The majority of instructions would be related to arithmetic and control operations to allow each program to flow correctly, with a medium amount dedicated to memory access, and only a select few necessary for system calls. In total, I estimate around 50 instructions would be necessary to completely cover any necessary application. The full list can be found at the bottom in Appendix A (from Assignment 2)
     - 1.1.4. As previously stated, the instruction size is 5 bits, which allows for 32 instructions per Type. This allows for not only a large library of instructions, but also to keep the four Types in mind. The thought process was to make it easy to program and debug by first focusing on the Type code, and then digging into the instruction. The downside is that 5 bit forces the total word to be an awkward 19 bits in length. Ideally the instruction would be four bits to limit the size down to a more "standard" 18 bits, but that would only allow for 16 instructions per type (or 64 total possible instructions). More flexibility and ease of programming outweighed the need for smallest size possible

   - 1.2. Instruction type and format
     - 1.2.1. As previously stated, REOM will use 4 instruction types: arithmetic, branch/control, memory access, and system calls. Each will be identified by the first two bits of the instruction word.
     - 1.2.2. Similar MIPS, the format for each word will change based on the Type. For both Type 11 and 01 (arithmetic and branch/control, respectively), the word will be as follows: 2 bits for the type, 5 bits for the OpCode, and 4 bits for each of the three registers or constant available. Like MIPS, the first register will be the destination register, while the second and third are the ones to perform the operation on. In the event of an addi (or similar), the last space will be used for any constants. While 5 bits only allows for constants up to 32 (2 raised to the power of 5), this is not a huge issue. I do not intend for large constants to be added for practical implementations, so there is no real need to extend this higher. Simple loops stepping by 1, multiples of 2, etc. are all supported in this current structure.

Next, Type 10 (memory access) will be as follows: 2 bits for the type, 5 bits for the OpCode, 4 bits for the register to either load or store from (including the derivations of load and store), leaving the last 8 bits for memory addresses. With 8 bits for memory, that allows for 256 different memory locations. As we will see in subsequent sections, the need should not be that high, but it is at least available.

Finally, Type 00 (system calls) will be as follows: 2 bits for the type, 5 for the OpCode, and the remaining 12 bits for miscellaneous or meta-data. The nature of system calls varies substantially, with some requiring very little information (such as exit) while others require more advanced input (such as a print statement). As such, it is important to allow for a wide variety of information for these instruction Types.

1.3. Processor architecture and characteristics

1.3.1. As previously mentioned, REOM will use a single coprocessor setup. This is a good balance between cost and efficiency. The coprocessor arrangement allows for some flexibility in allowing for floating point arithmetic which is about all the complexity this system would need. Adding an additional multicore processor would increase the cost an unnecessary amount. In most MIPS applications, there are generally a second set of coprocessors that allow for application-specific operations. This makes MIPS far more versatile and able to handle many more tasks than the REOM arrangement. However, the two ISA's were developed for two different reasons. The REOM implementation will be less costly and physically smaller, allowing for perhaps even more specialized applications.

No multithread will be supported as concurrent processes will not to be ran. Allowing for simultaneous execution is a huge benefit in certain applications, such as operating systems, but REOM is not intended for any such circumstance. Thus, no benefit would be derived from increasing overall throughput, and in fact it will be quite uncommon to have more than one program executed at a time. To allow for multithreading requires additional hardware overhead, which again goes against the core design philosophy.

1.3.2. While multithreading may be overkill, pipelining is certainly a requirement. Within a single process, there are multiple operations which must be processed, registers accessed, and data stored. Given that the architecture utilizes a single coprocessor, maximizing its workload time is quite important. Even though there are some different instructions, REOM will make use of the same five-stage process as MIPS: instruction fetch, instruction decode, execution (also performed on the ALU), memory cycle, and write back (register update). This five-stage model performs well and there is no reason to alter it further.

Unfortunately, this approach also comes with the issue of hazards between instructions. When multiple instructions rely on the same register, a potential issue is created. There are different types of hazards, and different ways to identify and deal with each. In the case of REOM, it will not have a forwarding unit or any other dedicated piece of hardware. As such, the only option will be to introduce a "stall" cycle, which is an empty process (or processes) for one of the instructions with the dependency. In some applications where user response time is critical, this is not a practical solution and more hardware would be appropriate. Here though, stalls should be infrequent (given the cache design, discussed later) and better program design can also be used to further reduce the likelihood of dependencies.

2. Identify the memory structure for your proposed architecture

2.1. The memory structure will be similar to MIPS in that it will be byte-addressable. Given that there are 8 bits available for memory location, this is a natural fit. To focus on simplicity, memory will be unified for both instruction and data. While there are some benefits to separation, the extra hardware needed to simply access each (a second bus), there needs to be extra instructions and timing checks to ensure that no out of order execution occurs: the correct instruction must be paired up with the correct data.

Additionally, because REOM is using both pipelining and a two-level cache system (see section 2.4 below), there may in fact not be any performance gain with using a separate memory for each. If the pipelining is robust enough to handle hazards and the cache system is advanced enough, there is no need for separate memories.

2.2. REOM can support up to 16 addresses, resulting in 65MB (2^{16}) of memory. This is less than MIPS, but again, the intended purpose is different. There is not any reason to need more for a small, fast system like this one. The trade-off is that, in the even the user needs more space, there are no additional options for the programmer.

2.3. One of the four Types of instruction groups in REOM is dedicated to memory access. The commands are similar to MIPS with the concept of both a load and store concept. There are specific commands to load and store memory locations. Specifically, the memory (data segment) address could be stored in an array, and the elements in that location could be accessed after that location is loaded into a register. Additionally, there are other commands to load and store arrays at a register, and the register will hold the memory location. To the user, it looks like he or she is accessing an element at a register, but they are actually locating a specific place in memory that is “hidden” from them. In the event that an array is stored using this method, the next available memory location will be used.

2.4. REOM will have split Level 1 caches for instructions and data, and then a unified Level 2 cache for both. The reason is that, because we are using stalls instead of a dedicated piece of hardware (such as a forwarding unit) to deal with hazards, it is important to minimize the likelihood of a dependency situation. The separate Level 1 caches allow for the potential avoidance of a structural hazard (but only at this level). Additionally, the average memory access time should be at least as good as a unified cache. The size of the Level 1 cache will be 64KB (2^6) **each**, and the Level 2 cache will simply be a single 64KB cache. Here the need for a bigger Level 1 cache is to really focus on having as many hits in the first level as possible. The system works best by minimizing trips to Level 2 and would be penalized (in terms of speed) by having to access it. Conversely, not having a larger Level 2 cache saves a little in terms of both cost and size. Even though Level 1 cache is more expensive, it is worthwhile to invest there and structure the system so Level 2 relied upon as little as possible (before going to memory).

2.5. With the heavy reliance on the cache, REOM will use a fully associative structure. While this method has the highest cost, it also has the best performance. We want the cache here to be as fast as possible and will concede a bit of cost for the extra performance. Blocks are found using a simple block offset method, where the block address is supplemented by an offset. Each block frame in the cache has a unique address in memory, and a single bit is included to determine if it is valid or not (1 or 0, respectively). On a miss ideally, we would use a more sophisticated approach like least recently used (LRU), but as we have already taken advantage of a fully associative cache, we will use a random block replacement method. Fortunately, this design does allow for uniform allocation (which should benefit the fully associative structure). Finally, on a write, we will use a write back approach. When there is a miss, the data is only written into the blocks of the cache level. This minimizes memory and bus traffic; however, it comes at the cost of consistency. The desire to save cost outweighs the benefit of a write through approach. Even though we may experience write stalls, because the majority of all instruction access are reads, this is a cost-benefit tradeoff that is acceptable.

2.6. Because the system has a shared memory system with a coprocessor, as well as a separate memory cache for both instruction and data, it is possible to have multiple copies of a data value. As such, a system must be put into place to prevent outdated data to be acted upon. One of the goals of REOM is to minimize hardware additions, both for cost and size, however in this case it is preferable to add a proper coherency controller. Given the multiple registers, cache structure, and unified memory, the risk of error is quite high. Also, the heavy reliance on the caches (specifically Level 1) necessitates that this particular part of the system runs efficiently. Additionally, because there are not a lot of blocks (or nodes) to update, this implementation will work quite well. Had the system been more complicated, updating every request to every node would become impractical. In this case, and again because we are using a write back cache, REOM uses a write update method. That is, when a write is made, all copies are updated via the bus. We assume that the majority of memory accesses are reads which implies that after a write, the next several accesses are reads. In this case, it is beneficial to have the block updated and ready for all potential processes/instructions that may need to read from it next. This is less efficient than a write invalidate system where, upon a processor write, copies of the block are not immediately updated but rather marked as invalidated. Only once they are needed is the validation checked.

There are only three states in this system: exclusive, shared, or modified. Exclusive means the cache block is valid and only held in one place. Thus, it presents no issues. Shared indicates the cache block is held in more than one cache but is still valid. Finally, modified indicates that the block is the only copy of the memory AND it has been modified since being retrieved from memory. In this state, a write back will occur when a block is replaced. There is no need for an invalidated state as there would be with a write invalidate system.

3. Appendix A:

Description	Item	Type	Number	Code	OpCode
add	add	11	1	00001	1100001
add immediately	addi	11	2	00010	1100010
subtract	sub	11	3	00011	1100011
subtract immediately	subi	11	4	00100	1100100
multiply	mult	11	5	00101	1100101
divide	div	11	6	00110	1100110

modulo	mod	11	7	00111	1100111
power	pow	11	8	01000	1101000
square root	sqrt	11	9	01001	1101001
square root of number times pi	sqpi	11	10	01010	1101010
bitwise and	and	11	11	01011	1101011
bitwise or	or	11	12	01100	1101100
bitwise xor	xor	11	13	01101	1101101
bitwise shift left	bitl	11	14	01110	1101110
bitwise shift right	bitr	11	15	01111	1101111
multiply immediately	multi	11	16	10000	1110000
divide immediately	divi	11	17	10001	1110001
modulo immediately	modi	11	18	10010	1110010
power immediately	powi	11	19	10011	1110011
square root immediately	sqrti	11	20	10100	1110100
square root of number times pi immediately	sqpii	11	21	10101	1110101
bitwise and immediately	anii	11	22	10110	1110110
bitwise or immediately	orii	11	23	10111	1110111
bitwise xor immediately	xori	11	24	11000	1111000
bitwise shift left immediately	bilti	11	25	11001	1111001
bitwise shift right immediately	bitri	11	26	11010	1111010
		11	27	11011	1111011
		11	28	11100	1111100
		11	29	11101	1111101
		11	30	11110	1111110
		11	31	11111	1111111
equal to	eqto	01	1	00001	100001
graeter than	gret	01	2	00010	100010
greater than or equl to	greq	01	3	00011	100011
less than	lest	01	4	00100	100100
less than or equal to	lesq	01	5	00101	100101
not equal	noeq	01	6	00110	100110
equal to immediately	eqtoi	01	7	00111	100111
greater than immediately	greti	01	8	01000	101000
greater than or equal to immediately	greqi	01	9	01001	101001
less than immediately	lesti	01	10	01010	101010
less than or equal to immediately	lesqi	01	11	01011	101011
not equal immediately	noeqi	01	12	01100	101100
jump to a specific input line	jmpl	01	13	01101	101101
jump to a defined instruction	jmpi	01	14	01110	101110
comment ignore	#	01	15	01111	101111
		01	16	10000	110000
		10	1	00001	1000001
store array	sarr	10	2	00010	1000010

load array	larr	10	3	00011	1000011
load item from word	lw	10	4	00100	1000100
store item in word	sw	10	5	00101	1000101
		10	6	00110	1000110
		10	7	00111	1000111
		10	8	01000	1001000
		10	9	01001	1001001
		10	10	01010	1001010
		10	11	01011	1001011
		10	12	01100	1001100
		10	13	01101	1001101
		10	14	01110	1001110
		10	15	01111	1001111
		10	16	10000	1010000
terminate gracefully	exit	00	1	00001	000001
print text	printt	00	2	00010	000010
print register direct	printr	00	3	00011	000011
		00	4	00100	000100
		00	5	00101	000101
		00	6	00110	000110
		00	7	00111	000111
		00	8	01000	001000
		00	9	01001	001001
		00	10	01010	001010
		00	11	01011	001011
		00	12	01100	001100
		00	13	01101	001101
		00	14	01110	001110
		00	15	01111	001111
		00	16	10000	010000