Group 2 YMC-to-Binary Encoding

Details

- Instructions have variable-length arguments, so two different instructions can take up a different amount of space in memory.
    - Since we have 3-argument 8-bit arithmetic operations required, every instruction would have to be 4 bytes wide with fixed length arguments. However, some instructions only need one or two bytes to implement. Fixed lengths would simplify simulation, but waste significant space.
- Arguments are one-byte, with the exception of register-register operations, which are one-byte for both registers. WIth 4 general purpose registers, each can easily be assigned to 4 bits.
    - EAX is 0001, EBX is 0010, ECX is 0100, and EDX is 1000. The higher-order 4 bits, those on the left of the byte, will be the first argument, and the lower-order bits will be the second argument.
- Arithmetic will have Register-register implementations, as well as register-memory implementations. For 3-argument arithmetic, we will have R-R-R only. Since 3-argument arithmetic generates 12 necessary instructions guaranteed, adding more can increase workload heavily. When converting from HLC, this should be EAX, EBX, and ECX. EDX should be unchanged. All arithmetic will be in the form add x y, which executes x = x + y. The first argument should be on the left in the math process, which matters for subtraction and division.
- Multiplication and Division will have signed and unsigned versions. Addition and subtraction work the same for signed and unsigned numbers. Given this complication, we need 10 additional 3-argument arithmetic operations, bringing the total to 22. There is no three-operand instruction to do signed multiplication -> unsigned division, or any similar mismatch.
- Jump operations use flags set by subtract or cmp instructions, mov moves the value of the second operand into the destination of the first operand.

Implementation

- We will have 3 mov instructions. movrr, movrm, movmr and movrl. Mov register register, mov register memory, mov memory register and mov register literal. We can’t move directly from memory to memory, or literal to memory.
- We will have 12 two-operand arithmetic operations. add, sub, mult, smult, div, sdiv. The s signifies a signed operation, these all operate register-register

arithmetic. Additionally, addrm, subrm, multrm, smultrm, divrm, and sdivrm will represent arithmetic from register to memory.

- We have two comparison operations, cmprr, and cmprm. These perform a subtraction without storing the result, setting the same flags as subtraction to be used by jump instructions below.
- We have 7 jump operations, 6 conditional and one unconditional. jg jumps if the sign flag AND zero flags are 0. jge jumps if the sign flag is 0. jl jumps if the sign flag is 1, jle jumps if the sign flag OR the zero flag is 1. jne jumps if the zero flag is 0. je jumps if the zero flag is 1. jmp jumps regardless. All of these are one-argument instructions, taking the address of the instruction to jump to. However, their argument will be 2-bytes. We have 1kb of available RAM, so addresses need to be stored as 2-bytes.
- We have 26 three-operand instructions, described in the details section. These will need 2 bytes for arguments, one storing two registers and another byte storing the third.
- We have 5 categories of instruction, which we can fit into 3 bits. Our largest category has 26 instructions, which we can fit into 5 bits. Our first 3 bits will represent the category described above, 000 for mov, 001 for two-operand arithmetic, 010 for comparison, 011 for jumps, and 100 for three-operand arithmetic.
- We will also store a map of argument widths with each instruction in the simulation. Register operations will have one-byte, even for 2 arguments, as described above under the details section. The longest instructions will be 4-bytes, Anything that implements a register-memory operation. We need one-byte for the register, and 2 bytes for the memory. Technically, we could use 4-bits for the register and 12 bits for the memory address, but that complicates things to a degree that I don't think it would be worth it.
- Additionally, we have a single halt command, implemented with 0x00. This technically puts it in the mov category, but it will be handled as a special case.

Complete Byte Table

Category	Instruction	Total Width	Binary
Halt	hlt	1	0x00
Move	movrr	2	0x01
Move	movrm	4	0x02

Category	Instruction	Total Width	Binary
Halt	hlt	1	0x00
Move	movrl	3	0x23
2-arg Arithmetic	add	2	0x20
2-arg Arithmetic	sub	2	0x21
2-arg Arithmetic	mul	2	0x22
2-arg Arithmetic	smul	2	0x23
2-arg Arithmetic	div	2	0x24
2-arg Arithmetic	sdiv	2	0x25
2-arg Arithmetic	addrm	4	0x26
2-arg Arithmetic	subrm	4	0x27
2-arg Arithmetic	mulrm	4	0x28
2-arg Arithmetic	smulrm	4	0x29
2-arg Arithmetic	divrm	4	0x2A
2-arg Arithmetic	sdivrm	4	0x2B
Comparison	cmprr	2	0x40
Comparison	cmprm	4	0x41
Jump	jmp	3	0x60
Jump	jg	3	0x61
Jump	jge	3	0x62
Jump	jl	3	0x63
Jump	jle	3	0x64
Jump	jne	3	0x65
Jump	je	3	0x66
3-arg Arithmetic	addsub	3	0x80

Category	Instruction	Total Width	Binary
Halt	hlt	1	0x00
3-arg Arithmetic	addmul	3	0x81
3-arg Arithmetic	addsmul	3	0x82
3-arg Arithmetic	adddiv	3	0x83
3-arg Arithmetic	addsdiv	3	0x84
3-arg Arithmetic	subadd	3	0x85
3-arg Arithmetic	submul	3	0x86
3-arg Arithmetic	subsmul	3	0x87
3-arg Arithmetic	subdiv	3	0x88
3-arg Arithmetic	subsdiv	3	0x89
3-arg Arithmetic	muladd	3	0x8A
3-arg Arithmetic	mulsub	3	0x8B
3-arg Arithmetic	muldiv	3	0x8C
3-arg Arithmetic	smuladd	3	0x8D
3-arg Arithmetic	smulsub	3	0x8E
3-arg Arithmetic	smulsdiv	3	0x8F
3-arg Arithmetic	divadd	3	0x90
3-arg Arithmetic	divsub	3	0x91
3-arg Arithmetic	divmul	3	0x92
3-arg Arithmetic	sdivadd	3	0x93
3-arg Arithmetic	sdivsub	3	0x94
3-arg Arithmetic	sdivsmul	3	0x95