COMPUTER ARCHITECTURE REVIEW

ARDEN RASMUSSEN

MARCH 14, 2019

1. Representing and Manipulating Information

1.1. **Information Storage.** Computers store data in blocks of 8bits or *bytes*. These bytes are represented as a large array, where each element can be indexed individually.

1.1.1. *Hexadecimal Notation.* Instead of representing numbers from 0-9 hex represents them as 0-F. Its pretty straight forward

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Converting from decimal to hex is done by the process outlined below

$$\begin{aligned} x &= 9430 \\ \frac{9430}{16} &= 589r6 \\ \frac{589}{16} &= 36r13 \\ \frac{32}{16} &= 2r4 \\ \frac{2}{16} &= 0r2 \\ x &= 0 \times 24\text{D}6 \end{aligned}$$

1.1.2. *Data Sizes.* The index of the memory is represented by a *word*, thus the size of the memory is limited by the size of the word representation. Addresses can range from $0 - 2^w - 1$, for a w-bit word size.

C declaration	Bytes
`char`	1
`short`	2
`int`	4
`long`	8
`char*`	8
`float`	4
`double`	8

1.1.3. *Addressing and Byte Ordering.* There is little and big endian. Little endian puts the most significant byte last, and the big endian puts the most significant byte first. Almost everything uses little endian.

1.1.4. *Representing Strings.* Text data almost universally utilizes ASCII, and thus is more platform independent than byte data.

1.1.5. *Representing Code.* Binary code is almost never compatible with another machine, or operating system. Thus most things must be compiled separately.

1.1.6. *Introduction to Boolean Algebra.*

not	`~`	`~p`
and	`&`	`p&q`
or	`|`	`p|q`
exclusive-or	`^`	`p^q`

1.1.7. *Bit-Level Operations in C.* This is just using the boolean algebra to implement bitwise manipulation. It can be fast and useful for some purposes.

1.1.8. *Logical Operations in C.*

and	`&&`	`p&&q`
or	`||`	`p||q`
not	`!`	`!p`

1.1.9. *Shift Operations in C.* There is right shift(>>) and left shift(<<). They in general move all the bits of some value by the provided count, then filling in the new bits with `0` most of the time. If you are using arithmetic right shift, then it will fill the new bits with a copy of the sign of that value. This is useful when it is necessary t preserve the sign of some number. Most implementations use arithmetic right shift for signed, and logical for unsigned , but not all, so be warned.

1.2. **Integer Representations.**

1.2.1. *Integral Data Types.* Signed numbers support negative values, and unsigned do not. This usually means that unsigned get an additional bit for the actual value, meaning that they can become much larger.

1.2.2. *Unsigned Encodings.* This is just as you would interpret a binary number.

$$\sum_{i=0}^{w-1} x_i 2^i$$

1.2.3. *Two's-Complement Encodings.* This is the method for storing signed integers, The idea is to have a sign bit at the beginning, then add the rest of the bits as an unsigned integer to the weight of that signed bit.

$$-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

1.2.4. *Conversions between Signed and Unsigned.* Both of these leave the bit representation unchanged and just change how it is interpreted.

Converting from two's complement to unsigned. For x such that $TMin_w \leq x \leq TMax_w$

$$\begin{cases} x + 2^w & x < 0 \\ x & x \geq 0 \end{cases}$$

Converting from unsigned to two's complement. For u such that $0 \leq u \leq UMax_w$

$$\begin{cases} u & u \leq TMax_w \\ y - 2^w & u > TMax_w \end{cases}$$

1.2.5. *Signed versus Unsigned in C.* C allows for both signed and unsigned, and can mostly handle the conversations between the two but there are some cases when issues will happen. For example `-1 < 0U` $\rightarrow$ `0`. Because the bit representation of `-1` is a very large number if it were to be interpreted as an unsigned integer.

1.2.6. *Expanding the Bit Representation of a Number.* Converting from an unsigned number to a larger data type, we simply add leading zeros. Converting a two's complement number to a larger data type, we simply add leading copies of the most significant bit.

1.2.7. *Truncating Numbers.* Truncation of an unsigned number is just done by cutting off all of the bits that don't fit into the new representation.

Truncation of a two's complement number is a similar practice, but also copying the sign bit to the new most significant bit.

1.3. **Integer Arithmetic.**

1.3.1. *Unsigned Addition.* For x and y such that $0 \leq x, y < 2^w$

$$x +_w^u y = \begin{cases} x + y & x + y < 2^w \text{ normal} \\ x + y - 2^w & 2^w \leq x + y < 2^{w+1} \text{ overflow} \end{cases}$$

Unsigned negation for any number x such that $0 \leq x < 2^w$, its w-bit unsigned negation $-_w^u x$ is given as

$$-_w^u x = \begin{cases} x & x = 0 \\ 2^w - x & x > 0 \end{cases}$$

1.3.2. *Two's-Complement Addition.* For integer values x and y in range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$

$$x +_w^t y = \begin{cases} x + y - 2^w & 2^{w-1} \leq x + y \text{ positive overflow} \\ x + y & -2^{w-1} \leq x + y < w^{w-1} \text{ normal} \\ x + y + 2^w & x + y < -2^{w-1} \text{ negative overflow} \end{cases}$$

1.3.3. *Two's-Complement Negation.* For x in the range $TMin_w \leq x \leq TMax_w$, its two's complement negation $-_w^t x$ is given as

$$-_w^t x = \begin{cases} TMin_w & x = TMin_w \\ -x & x > TMin_w \end{cases}$$

1.3.4. *Unsigned Multiplication.* For x and y such that $0 \leq x, y \leq UMax_w$

$$x *_w^u y = (x \cdot y) \mod 2^w$$

1.3.5. *Two's-Complement Multiplication.* For x and y such that $TMin_w \leq x, y \leq TMax_w$

$$x *_w^t y = U2T_w((x \cdot y) \mod 2^w)$$

Basically do the unsigned multiplication and then convert to a signed integer.

1.3.6. *Multiplying by Constants.* Multiplying by powers of 2 is the same as shifting the bits to the left by that power.

1.3.7. *Dividing by Power of 2.* Dividing by a power of two is just shifting the bits by that power to the right.

1.4. **Floating Point.**

1.4.1. *IEEE Floating-Point Representation.* The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$.

- The sign s determines weather the number is negative or positive, where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The significand M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The exponent E weights the value by a (possibly negative) power of 2.

This splits the bits into three blocks, the sign bit. The k-bit exponent, and the n-bit fraction.

Normalized Values. This is when the exponent is not all zeros and not all ones. When this is the case the value of the exponent $E = e - Bias$ where e is the unsigned number represented by the bits, and $Bias = 2^{k-1} - 1$. The fraction field is interpreted as representing the fractional value f, where $0 \leq f < 1$, then the significand is defined as $M = 1 + f$

Denormalized Values. When the exponent field is all zeros, then $E = 1 - Bias$, and $M = f$

Special Values. If the exponent is all ones, and the fraction is all zeros, then this is ∞, if the fraction is not all zeros, then the value is Nan.

1.4.2. *Rounding.* There are four methods of rounding. Round-to-event, Round-toward-zero, Round-down, Round-up.

1.4.3. *Floating Point in C.* C provides `float` and `double`, and uses round-to-event. And it will appropriately handle the conversion between integers and floating point with possible rounding or overflow appropriately.

2. Machine-Level Representation of Programs

2.1. **Program Encodings.** The compiler takes the C code through several levels of abstraction before it can generate the executable.

2.1.1. *Machine-Level Code.* The machine level code is dependent on the instruction set architecture. These ISAs describe how the program behaves for every given instruction. The compiler converts C into assembly code. Assembly code is just one step up from the binary, but is much more readable.

2.1.2. *Notes on Formatting.* Add comments, it makes things easier.

2.2. **Data Formats.**

C	Intel	Assembly Suffix	Bytes
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double Word	`l`	4
`long`	Quad Word	`q`	8
`char`*	Quad Word	`q`	8
`float`	Single Precision	`s`	4
`double`	Double Precision	`l`	8

2.3. **Access Information.** Registers are used to temporally store values, there are names for accessing only subsections of a register, but it all stores the same information.

63	31	15	7	Type
`rax`	`eax`	`ax`	`al`	Return value
`rbx`	`ebx`	`bx`	`bl`	Callee saved
`rcx`	`ecx`	`cx`	`cl`	4th argument
`rdx`	`edx`	`dx`	`dl`	3rd argument
`rsi`	`esi`	`si`	`sil`	2nd argument
`rdi`	`edi`	`di`	`dil`	1st argument
`rbp`	`ebp`	`bp`	`bpl`	Callee saved
`rsp`	`esp`	`sp`	`spl`	Stack pointer
`r8`	`r8d`	`r8w`	`r8b`	5th argument
`r9`	`r9d`	`r9w`	`r9b`	6th argument
`r10`	`r10d`	`r10w`	`r10b`	Caller saved
`r11`	`r11d`	`r11w`	`r11b`	Caller saved
`r12`	`r12d`	`r12w`	`r12b`	Callee saved
`r13`	`r13d`	`r13w`	`r13b`	Callee saved
`r14`	`r14d`	`r14w`	`r14b`	Callee saved
`r15`	`r15d`	`r15w`	`r15b`	Callee saved

2.3.1. *Operand Specifiers.*

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + Displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled Indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled Indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled Indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled Indexed

2.3.2. *Data Movement Instructions.*

Instruction	Effect		Description
`MOV`	S, D	$D \leftarrow S$	Move
`movb`			Move Byte
`movw`			Move Word
`movl`			Move Double Word
`movq`			Move Quad Word
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word
`MOVZ`	S, R	$R \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
`movzbw`			Move zero-extended byte to word
`movzbl`			Move zero-extended byte to double word
`movzbq`			Move zero-extended byte to quad word
`movzwl`			Move zero-extended word to double word
`movzwq`			Move zero-extended word to quad word
`movzlq`			Move zero-extended double word to quad word
`MOVS`	S, R	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
`movsbw`			Move sign-extended byte to word
`movsbl`			Move sign-extended byte to double word
`movsbq`			Move sign-extended byte to quad word
`movswl`			Move sign-extended word to double word
`movswq`			Move sign-extended word to quad word
`movslq`			Move sign-extended double word to quad word

2.3.3. *Pushing and Popping Stack Data.*

Instruction	Effect	Description
`pushq` S	$R[\%rsp] \leftarrow R[\%rsp] - 8$	Push quad word
	$M[R[\%rsp]] \leftarrow S$	
`popq` D	$D \leftarrow M[R[\%rsp]]$	Pop quad word
	$R[\%rsp] \leftarrow R[\%rsp] + 8$	

2.4. **Arithmetic and Logical Operations.**

Instruction	Effect	Description
`leaq` S, D	$D \leftarrow \&S$	Load effective address
`INC` D	$D \leftarrow D + 1$	Increment
`DEC` D	$D \leftarrow D - 1$	Decrement
`Neg` D	$D \leftarrow -D$	Negate
`Not` D	$D \leftarrow \ D$	Complement
`ADD` S, D	$D \leftarrow D + S$	Add
`SUB` S, D	$D \leftarrow D - S$	Subtract
`IMUL` S, D	$D \leftarrow D * S$	Multiply
`XOR` S, D	$D \leftarrow D \wedge S$	Exclusive-or
`OR` S, D	$D \leftarrow D \vert S$	Or
`AND` S, D	$D \leftarrow D\&S$	And
`SAL` k, D	$D \leftarrow D << k$	Left shift
`SHL` k, D	$D \leftarrow D << k$	Left shift
`SAR` k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
`SHR` k, D	$D \leftarrow D >>_L k$	Logical right shift

2.5. **Special Arithmetic Operations.**

Instruction	Effect	Description
`imulq` S	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
`mulq` S	$R[\%rdx] : R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply
`cqto`	$R[\%rdx] : R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
`idivq` S	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \mod S$	Signed divide
	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \div S$	
`divq` S	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \mod S$	Unsigned divide
	$R[\%rdx] \leftarrow R[\%rdx] : R[\%rax] \div S$	

2.6. **Control.**

2.6.1. *Condition Codes.*

`CF:` Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
`ZF:` Zero flag. The most recent operation yielded zero.
`SF:` Sign flag. The most recent operation yielded a negative value.
`OF:` Overflow flag. The most recent operation caused a two's complement overflow–either negative or positive.

Note that the `leaq` operation does not alter condition codes, but all other arithmetic operations do.

There are also two instructions that set the condition codes without altering the registers.

Instruction	Based on		Description
`CMP`	S_1, S_2	$S_2 - S_1$	Compare
`TEST`	S_1, S_2	$S_1 \& S_2$	Test

2.6.2. *Accessing the Condition Codes.* The `SET` instruction is used to get the values stored in the condition codes.

Instruction	Synonym	Effect	Set condition
`sete` *D*	`setz`	$D \leftarrow ZF$	Equal/Zero
`setne` *D*	`setnz`	$D \leftarrow\ ZF$	Note Equal/Not Zero
`sets` *D*		$D \leftarrow SF$	Negative
`setns` *D*		$D \leftarrow\ SF$	Nonnegative
`setg` *D*	`setnle`	$D \leftarrow\ (SF \wedge OF) \&\ ZF$	Greater (signed $>$)
`setge` *D*	`setnl`	$D \leftarrow\ (SF \wedge OF)$	Greater or equal (signed $\geq$)
`setl` *D*	`setnge`	$D \leftarrow SF \wedge OF$	Less (signed $<$)
`setle` *D*	`setng`	$D \leftarrow (SF \wedge OF) \vert ZF$	Less or equal (signed $\leq$)
`seta` *D*	`setnbe`	$D \leftarrow\ CF \&\ ZF$	Above (unsigned $>$)
`setae` *D*	`setnb`	$D \leftarrow\ CF$	Above or equal (unsigned $\geq$)
`setb` *D*	`setnae`	$D \leftarrow CF$	Below (unsigned $<$)
`setbe` *D*	`setna`	$D \leftarrow CF \vert ZF$	Below or equal (unsigned $\leq$)

2.6.3. *Jump Instructions.* The jump instruction causes the program execution to change to a new part of the code.

Instruction	Synonym	Jump condition	Description
`jmp` *Label*		1	Direct Jump
`jmp` $*Operand$		1	Indirect Jump
`je` *Label*	`jz`	`ZF`	Equal/Zero
`jne` *Label*	`jnz`	`~ZF`	Not Equal/Not Zero
`js` *Label*		`SF`	Negative
`jns` *Label*		`~SF`	Nonegative
`jg` *Label*	`jnle`	$\ (SF \wedge OF) \&\ ZF$	Greater (signed $>$)
`jge` *Label*	`jnl`	$\ (SF \wedge OF)$	Greater or equal (signed $\geq$)
`jl` *Label*	`jnge`	$SF \wedge OF$	Less (signed $<$)
`jle` *Label*	`jng`	$(SF \wedge OF) \vert ZF$	Less or equal (signed $\leq$)
`ja` *Label*	`jnbe`	$\ CF \&\ ZF$	Above (unsigned $>$)
`jae` *Label*	`jnb`	$\ CF$	Above or equal (unsigned $\geq$)
`jb` *Label*	`jnae`	CF	Below (unsigned $<$)
`jbe` *Label*	`jna`	$CF \vert ZF$	Below or equal (unsigned $\leq$)

2.6.4. *Jump Instruction Encodings.* Most jump instructions use relative jumps, they jump based upon the difference between the target and the instruction following the jump. Then there is the absolute, where the address is the absolute position of the next instruction.

2.6.5. *Implementing Conditional Branches with Conditional Control.* The most general way of implementing conditional code in assembly is to use conditional jumps. If some condition has been met, then the jump is made otherwise the jump is not executed.

2.6.6. *Implementing Conditional Branches with Conditional Moves.* This method is to compute both results of some conditional, then depending of the result of the condition save that value. This is very efficient, with the exception of when there is a lot of computation in the condition.

2.6.7. *Loops.*
Do-While Loops. This executes the statement, evaluates the test expression and continues the loop if the evaluation result is nonzero.

While Loops. This must evaluate the test expression first, then if the evaluation result is nonzero, then it enters the loop. Once it enters the loop it acts the exact same as a do-while loop.

2.6.8. *Switch Statements.* These are great. They are often implement through the use of a jump table. This is a table where the entry i is the address of the instruction that should be preformed when the switch index is equal to i. Thus there are no conditional statements that are being evaluated, there is only jumping to a position based upon the value of the index.

2.7. **Procedures.** There are three main parts of a procedure. Passing control, Passing data, and Allocating and deallocating memory.

2.7.1. *The Run-Time Stack.* A *stack frame* is a block on the stack that is specifically allocated for each procedure. The stack frame is the section of the stack that stores any additional local variable, the arguments, the saved registers, and the return address for the callee function.

2.7.2. *Control Transfer.* Passing control involves simply setting the program counter to the starting address for the code of the procedure.

Instruction	Description
`call` *Label*	Procedure call
`call` $*$*Operand*	Procedure call
`ret`	Return from call

2.7.3. *Data Transfer.* Argument data is passed through registers, and the return value is stored in `rax`. If there are more than six arguments then the additional ones are passed on the stack.

2.7.4. *Local Storage on the Stack.* Typically local variables are stored in the stack frame if there is not enough registers to handle all of them.

2.7.5. *Local Storage in Registers.* The registers are a constant source of data, and are accessible by any procedure. Thus we need to save the values in the register such that the called function does not disrupt the values that the callee procedure has stored in those registers. `rbx`, `rbp`, and `r12-r15`, these is called callee-saved registers, because it is up to the callee to preserve the values of the registers. All other registers are caller saved.

2.7.6. *Recursive Procedures.* Recursive calls is handled by just creating a new stack frame for each recursive depth, such that one does not interfere with another.

2.8. **Array Allocation and Access.**

2.8.1. *Basic Principles.* Arrays are stored in contiguous lengths of memory. Using this the memory referencing instructions are designed to simplify array access.

2.8.2. *Pointer Arithmetic.* This is just explaining how pointers work, they are the exact same as arrays just with some slightly different notation.

2.8.3. *Nested Arrays.* Nested arrays are still stored contiguously in memory, meanings that the first row is immediately followed by the second and so on. This makes accessing the values from assembly very simple because we just dereference the value of $A[i][j] = A[W * i + j]$, where W is the width of the array.

2.8.4. *Fixed-Size Arrays.* The C compiler is able to make many optimizations on fixed size arrays, and so it is preferable to use these when possible.

2.8.5. *Variable-Size Arrays.* Variable sized arrays are not stored on the stack, because they must be done using `malloc`, so they are instead stored on the heap.

2.9. **Heterogeneous Data Structures.**

2.9.1. *Structures.* The assembly implementation of a `struct` is similar to that of an array. All values are stored in contiguous region of memory and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. To access elements, the compiler generates the appropriate offset to the address of that value in the structure.

2.9.2. *Unions.* `union` is a method to store any number of types in the same block of memory. This can greatly save on memory, if only one value is saved at a time. One important thing is to be able to interpret what type is currently stored in the union.

2.9.3. *Data Alignment.* Many computer systems require the data to be aligned and must be a multiple of some K (2,4,8). These requirements are for the sake of hardware. This alignment means that if dome data type takes less than the requirement then empty bytes are inserted in order to make it fill the specified alignment.