EECS 843 - Programming Language Foundation II

Perry Alexander, palexand@ku.edu

November 22, 2022

This document describes the instruction set architecture for the KURM09 that we will specify and verify for our final project. The KURM09 is similar to the RISC design used in a number of architecture texts, but is simpler to facilitate verification and compilation. This is a realistic CPU description that we used for instruction in EECS 443 for many years.

The KURM09 uses a Harvard-style architecture where instructions and data are stored in logically separate memories. Thus, it is impossible for an instruction read to access data memory or a data read or write to affect instruction memory. In reality, there is only one memory that is treated as two separate memories by a memory controller. The word length for both data and instruction memory is 16 bits. Although KURM09 reads and writes words, memory addresses index bytes. Table 1 shows the word in memory associated with each address.

Word	*Byte*	
0	0	Low
	1	High
1	2	Low
	3	High
2	4	Low
	5	High

Table 1: KURM09 memory organization.

Physical memory is accessed using byte addresses. However, both instruction fetch and data memory operations read and write words to memory. Thus, each memory access operates on two consecutive bytes. Because instructions are 16 bits in length, the program counter must be incremented by 2 to move to the next instruction. Bit 0 of each byte is least significant while bit 7 is most significant. Similarly, the low byte of each word is least significant and the high byte is most significant.

There are 16 registers available separated into a set of 14 orthogonal, general-purpose registers (R_2-R_{15}) and 2 constant value registers (R_0-R_1). R_0 always contains 0 and R_1 always contains 1. Neither R_0 nor R_1 can be modified by any instruction. Instructions specify register IDs using 4 bit values. Throughout this document, R_n refers to the register ID for register n.

The program counter is an internal 16-bit register that cannot be directly accessed by any instruction except jmpl. Every instruction increments the program counter by 2 with the exception of branch (bra) and jump (jmpl) that alter the program counter in different ways. Throughout this document, *PC* refers to the program counter.

The status register is an internal 4-bit register that cannot be directly accessed by any instruction. The status register is updated when an arithmetic or set instruction executes and is invariant otherwise. Assuming that X and Y are the first and second operands to an

instruction and $X \circ Y$ is the result of an instruction, the status register is updated according to table 2.

Bit	*Set Condition*
0	$X < Y$
1	$X = Y$
2	$X > Y$
3	$X \circ Y = 0$

Table 2: Bits defining the KURM09 status register.

KURM09 IS DEFINED BY THE INSTRUCTION SET defined in table 3. All instructions are one word in length with the format of each instruction shown in table 3. The high four bits always specify the operation while the low 12 bits specify registers, offsets, or masks, depending on the instruction type.

Instruction	*Meaning*	Op	R_s	R_t	R_d
add R_s,R_t,R_d	$R_d := R_s + R_t$	0000	0-15	0-15	0-15
sub R_s,R_t,R_d	$R_d := R_s - R_t$	0001	0-15	0-15	0-15
or R_s,R_t,R_d	$R_d := R_s \vee R_t$	0010	0-15	0-15	0-15
and R_s,R_t,R_d	$R_d := R_s \wedge R_t$	0011	0-15	0-15	0-15
set R_s,R_t,msk	set status $\wedge$ msk	0110	0-15	0-15	0-15
bra msk,off	if(msk $\wedge$ status) $\neq 0$ pc := pc + off	1110	0-15	off_{7-4}	off_{3-0}
lw R_s,R_t,off	$R_t := M(R_s + \text{off})$	0100	0-15	0-15	0-15
sw R_s,R_t,off	$M(R_s + \text{off}) := R_t$	0101	0-15	0-15	0-15
jmpl R_s,R_t,off	$R_t := PC + \text{off}$ $PC' := R_s$	0111	0-15	0-15	0-15

Table 3: KURM instruction set

Mathematical and logical operations (add, sub, and, or) operate on three registers. Data transfer, branch and jump operations (lw, sw, bra, jmpl) operate on two registers and a 4-bit absolute offset. The set operation operates on two registers and a 4-bit mask.

Mathematical and logical operations treat the low 12 bits as register identifiers. The high four bits represent R_d, the middle four R_s and the low four R_t as specified in table 3. Addition and subtraction (add, sub) treat the contents of R_s and R_t as 16 bit, two's compliment numbers. An overflow value should be generated by these instructions. Conjunction and disjunction (and, or) treat R_s, R_t and R_d as unsigned, 16 bit values. The only addressing mode used by arithmetic and logic operations is register direct.

The set (set) operation treats the contents of R_s and R_t as 16 bit, 2's compliment numbers. It performs four comparisons, $R_s = R_t = 0$, $R_s < R_t$, $R_s = R_t$ and $R_s > R_t$. The results of these comparisons are and'ed with the 4-bit mask value and stored in the four bits of the status registers as shown in table 4. The remaining high four bits are set to 0. As a psuedo-arithmetic instruction, the only addressing mode used by set is register direct.

7-4	3	2	1	0
0	$R_s = R_t = 0$	$R_s > R_t$	$R_s = R_t$	$R_s < R_t$

Table 4: Bit ordering in the status register for the set instruction

Load and store (lw, sw) operations use the low 12 bits to specify memory address, source/destination register and offset respectively. R_s specifies the register containing a base address. R_d specifies the

offset and R_t specifies the destination (or source) for data being read (or stored). Note that the only addressing mode is register indirect. The only addressing mode used by data transfer instructions is register indirect.

The branch (bra) instruction specifies a mask value and an 8-bit, 2's compliment word offset. The 4-bit mask value is bit-wise and'ed with the low four bits of the status register. If the result is non-zero, then the 8-bit offset is added to the program counter after converting to a byte offset value. The mask specifies the branch type by indicating status bits checked prior to branching. For example, "0001" specifies branch less than; "0011" specifies branch less than or equal; "1000" specifies branch 0; and "0101" specifies branch not equal.

The jump and link (jmpl) instruction specifies two registers and a 4-bit, 2's compliment word offset. When called, jmpl stores the current value of the PC plus the specified word offset value in R_t. Then, the value in R_s is loaded into the program counter. The objective of this instruction is to provide a branch mechanism that remembers where it branched from. If the address of a subroutine is stored in R_{15}, then jmpl $R_{15}, R_{14}, 1$ jumps to the address stored in R_{15} and stores the jump point plus 2 in R_{14}. When the subroutine is ready to return jmpl $R_{14}, R_k, 0$ will return to the instruction after the call point. The value of R_k is arbitrary as is the offset for a typical return.

Offsets for loading, storing, branching and jumping are 4 bit, 2's compliment numbers that specify offsets in words. Be cautious as you add and subtract offsets to get new program counter values. Further realize that the length of the offset limits how far a program can branch using the bra command.

Arithmetic, memory access, and jump commands have conditional equivalents that execute only if one of the low four bits of the status register is set. The conditional version of these instructions is specified by setting the high bit of the opcode. For example, the add operation uses opcode "0000" while the conditional add, addc uses opcode "1000". If none of the low four status bits are set when a conditional instruction executes, the instruction behaves like a no-op. Additionally, conditional instructions do not modify the low four bits of the status register. This will allow a multiple instructions to operate based on the same status register contents.

The example program in figure 1 shows a simple program that calls a subroutine adding two values twice. Memory location 0000 contains the address of a subroutine that is loaded into R_{15}. The jmpl instruction jumps to the address in R_{15} and stores the address from

the program counter plus 2 back into R_{15}.

In the subroutine (labeled R_{15} in the figure) R_{14} is used as a base address for obtaining two data values. These values are loaded into registers, added together and stored back into memory at the memory location in R_{14} plus 2 words. So, when the subroutine terminates, the original arguments are located in the two words at the memory location in R_{14} while the result is in the following memory loation.

The jmpl at the end of the subroutine returns to the address following the call site that was stored in R_{15}. The result value from the subroutine call is loaded into R_8. This value and the value in R_8 are stored at R_{14} to serve as arguments to the next subroutine call.

This example exhibits how the jmpl command is used to implement subroutine calls and how a register can be used to serve as a base address for finding arguments. Such operations are typical in code for RISC microprocessors like KURM09. However, RISC code like this is almost always generated by a compiler rather than written by hand. Furthermore, registers will be used instead of memory wherever possible to avoid the overhead associated with memory access.

```
        lw    R15, R0, 0
        jmpl  R15, R15, 1
        lw    R8, R14, 2
        sw    R8, R14, 0
        sw    R9, R14, 1
        jmpl  R15, R15, 1
        ...
R15 :   lw    R3, R14, 0
        lw    R4, R14, 1
        add   R3, R4, R5
        sw    R5, R14, 2
        jmpl  R15, R15, 0
```

Figure 1: Example program calling a subroutine twice.