

Exploring the Feasibility of a Robot-Centric CPU Simulator to Control Physical Robots as an Assembly Language Instructional Aid

by

Susan L. Gordon
slgordon@iusb.edu
Computer Science
Indiana University South Bend

James Wolfer
jwolfer@iusb.edu
Computer Science
Indiana University South Bend

Abstract: *In an effort to make program debugging more transparent, to provide an aspect of kinesthetic learning, and to inspire student interest, Indiana University South Bend incorporated robots into the assembly language curriculum. While the robots satisfied these original objectives, it is important for our students to acquire a deeper understanding of computer organization and its implication for program development. Toward this end, a custom, robot-centric, CPU instruction set was developed. Robot control programs written in this instruction set are then run on student-implemented simulators written in Intel assembly language. This study explores the feasibility of deploying a simulated CPU to control an actual robot. Specifically, we describe the simulated CPU, an implementation of an instruction set, and an initial assessment of the time penalties involved in the simulation with respect to controlling a robot.*

I. Introduction

Indiana University South Bend incorporated robots into its computer structures and assembly language curriculum in an effort to provide an engaging, kinesthetic environment for learning computer organization and assembly language [1]. While the robots satisfied these original objectives, they also introduced the side-effect that students tended to implement their robot control programs using a relatively restricted subset of the instructions and memory access modes available on a modern computer. This is a serious issue since the goals for this class include having students acquire a significant understanding of computer organization and its implication for overall program development rather than simply learning to program in assembly language. As a step toward the goal of deeper understanding, a custom, robot-centric instruction set was designed for subsequent implementation by students-written simulators. These simulators then execute student-developed assembly language programs that control actual robot behavior.

The application of both student-developed and instructor-provided CPU simulators as pedagogical tools has been a recurring theme in computer education. Examples include the MiniMIPS Simulation Project [2], the Harvard Ant-32 project [3], and the Minimal Instruction Set Computer [4]. The MiniMIPS project begins with template programs which students complete. Activities include parsing, simulating the functional units, and integrating datapath control. The authors report student feedback indicated that the project helped students to understand CPU organizational concepts. The Ant-32 project, in contrast, is provided to the students. The architectural goal was to present a clean, easy to understand processor which is functional enough to be used in courses in assembly languages, operating systems, and VLSI design. The authors of the Minimal Instruction Set Computer used a Java-based platform for simulating their machine instructions, for computer architecture exploration, and for operating systems. As with the Ant-32 system, this environment is supplied to the student.

While the systems surveyed provide working environments for students, adopting them would sacrifice the kinesthetic experience and student appeal of controlling robots both physically and under simulation. In contrast, this work describes the feasibility of simulating a custom, robot-specific, CPU instruction set for controlling both simulated and physical robots in real time on a typical, off-the-shelf computer. This, in turn, forms the basis for student-developed, simulated CPUs. Assembly language programs to control robots can then be developed for the simulated CPUs and tested on both simulated and physical robots.

Specifically, a custom CPU instruction set was developed and it was postulated that this instruction set could be used under simulation to control an actual robot on a modern computer. This would allow students to implement a “virtual CPU” on which they develop and test robot control programs. The robots provide a practical programming problem that can be tested on a robot simulator prior to controlling an actual robot. This combination of software simulation and robot interaction can provide both motivational and kinesthetic learning experiences to enhance student understanding of computer organization, to explore instruction operation, to understand memory access, and to gain additional practice in assembly language programming.

Before this custom CPU project could be deployed, an initial assessment of the time penalties involved in adding an additional layer of software to implement the CPU simulation was necessary. Deployment of an actual robot was also required to ensure that a physical robot could be controlled in real-time by programs written in the simulated instruction set. The resulting feasibility assessment included:

- A robot-specific instruction set.
- A virtual CPU to simulate instruction execution.
- Assessing the virtual CPU simulated instruction execution times.
- Using the virtual CPU to run a program written in the simulated instruction set to control a robot under simulation.
- Running a program written in the simulated instruction set to control a physical robot.

The balance of this report describes the implementation and results of this study, as well as an expanded instruction set designed to provide students experiences in implementing a wider variety of instructions and memory access modes ranging from simple logic to indirect jumps.

II. Hardware and Software

The hardware and software environment for this assessment consists of Khepera II robots and a customized simulator running under Linux. The K-Team Khepera II, shown in Figure 1, is a small, two-motor robot which uses differential motor speed for steering. It also features eight infrared proximity sensors positioned around the robot. As shown in the diagram in Figure 2, six sensors are positioned toward the “front” and two toward the “back” to provide sensory input values to detect obstacles [5].

As a software prototyping tool, a publicly available Khepera simulator, SIM [6], was modified. The SIM Khepera Simulator provides a workable subset of the Khepera robot command language. In this application, students write assembly language programs to communicate with and control the simulator and the robot through a pair of Linux pipes. The commands can be carried out in the simulated maze environment illustrated in Figure 2 or passed by the simulator to a physical Khepera robot through a serial port as shown in Figure 1. This combination provides the students with at-home testing ability in the simulated environment and supervised lab testing time to interact with the actual Khepera robots.

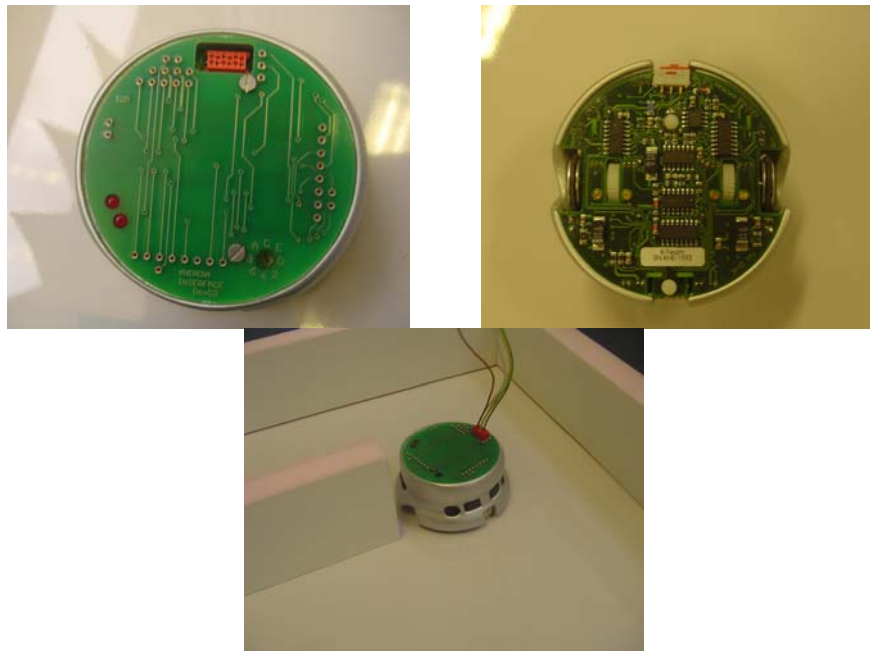


Figure 1: The K-Team Khepera II robot top view, bottom view, and shown in the test maze environment

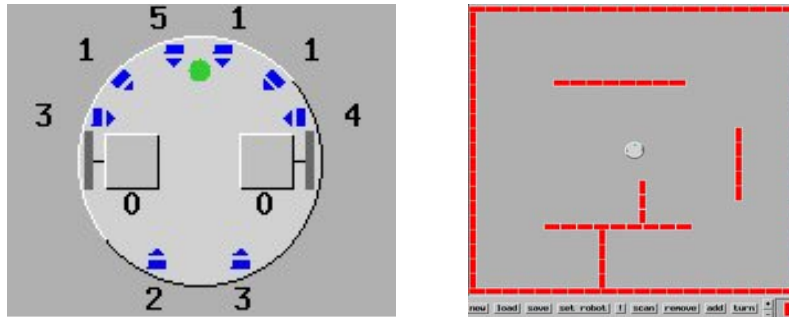


Figure 2: The SIM Simulator Khepera robot diagram and the simulator maze environment

III. Simulated CPU Organization

The virtual CPU for this feasibility study is organized with sixteen 32-bit registers, an instruction pointer, and a flag register along with an instruction set sufficient for writing programs to control a robot and for assessing simulator speed. This CPU organization is illustrated in Figure 3.

The instruction set was designed to include typical CPU instructions for data transfer, arithmetic, logic, and program control functions as shown in Figure 4. Move, load, and load immediate instructions were provided to get data into registers, and a store instruction moved register data back to memory. Arithmetic and logic instructions provided increment, decrement, and, or, exclusive or, and not functions for registers. A register to register compare instruction updated a flag register which could be tested by branch high, branch low, or branch if equal instructions. An unconditional jump instruction was also provided.

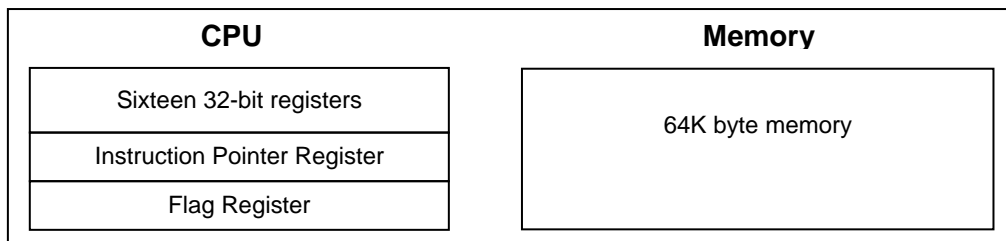


Figure 3: Virtual CPU organization

In addition to common CPU instructions, three custom instructions were provided for robot control. These instructions initialize communication with the Khepera robot or Khepera robot simulator, read robot sensor values, and set motor speeds.

```
# load instruction formats:
  mov  Rd,Rs          # register to register
  ldi  Rd,$immed     # immediate to register
  ld   Rd,memaddr    # memory address to register

# store instruction format:
  st   Rd,memaddr    # register to memory address
```

```

# logic and arithmetic instruction formats:
  and  Rd,Rs          # register to register
  or   Rd,Rs          # register to register
  eor  Rd,Rs          # register to register
  not  Rd              # register
  inc  Rd              # register
  dec  Rd              # register

# compare instruction format:
  cmp  Rd,Rs          # register to register – sets flag
# branch instruction formats:
  bg   location       # branch greater to location
  bl   location       # branch less to location
  beq  location       # branch equal to location

  jmp  location       # jump to location

# robot instruction formats:
  rinit                # initialize communication
  rsens Rd             # read robot sensors
  rspeed Rright,Rleft # set robot motor speeds

```

Figure 4: CPU instruction set

IV. Implementation of the Instruction Subset

The simulated CPU was implemented in Pentium assembly language using AT&T assembler syntax under Linux. The code used for this study is contained in the appendix with portions extracted for illustration in the figures below. The simulated CPU was allocated 64K of memory and sixteen 32-bit registers defined as arrays. The virtual CPU code to be interpreted by the simulator was assembled and placed into the 64K memory array. The Pentium edi register was reserved for the simulated instruction pointer, and register ecx was designated as the flag register which is referenced by the simulated CPU compare instruction.

The main loop in the CPU simulator carried out the classic fetch-decode-execute cycle as shown in Figure 5, with op-codes fetched from the memory array indexed by Pentium register edi. The decode-execute routines for individual instructions were reached through a dispatch table of pointers indexed by the operation field of the op-code. The individual instruction routines perform the appropriate operations and adjust the instruction pointer before returning to the main fetch-decode-execute loop.

```

# ***** fetch-decode-execute CPU cycle *****#
#
# AT&T assembly syntax is used.
# Register %edi is used as the simulated instruction pointer.
# Register %eax is a scratch register.
# The simulated 64K memory array is "smem" and
# "optbl" is the instruction dispatch table which is indexed by operation.

```

```

#
# *****#
-start:
    movl  $0,%edi                # Load the simulated ip with address zero
fetch:
    andl  0xffff,%edi           # Fetch the op code from the simulated memory
    movb  smem(,%edi,1),%al
    and   $0xff,%eax
decode:
    movl  optbl(,%eax,4),%eax    # Decode by operation dispatch table
    incl  %edi
execute:
    call  *%eax                 # Call the inst execution routine and store results
repeat:
    jmp   fetch                 # Go back and fetch the next instruction

```

Figure 5: The fetch-decode-execute CPU cycle

A sample routine implementing one virtual CPU instruction is shown in Figure 6. Specifically, this shows the implementation of the “mov Rd,Rs” instruction, which moves the contents of a source register, Rs, to a destination register, Rd. It is represented in memory as a two byte op-code with the first byte specifying the operation and the second byte encoding the source and destination registers.

```

# ***** mov Rd,Rs *****#
#
# AT&T assembly syntax is used.
# Move the contents of the source register Rs to the destination register Rd.
# The instruction pointer, %edi, points to the second byte of the op-code on entry.
# Registers %eax and %ebx are scratch registers.
# The simulated memory array is “smem” and “regs” is the array of simulated registers.
#
# *****#

    movb  smem(,%edi,1),%al      # get registers from instruction
    andl  $0xff,%eax
    movl  %eax,%ebx
    andl  $0x0f,%eax            # source register index to eax
    sarl  $4,%ebx               # destination register index to ebx
    movl  regs(,%eax,4),%eax     # get contents from register
    movl  %eax,regs(,%ebx,4)    # move contents to destination
    incl  %edi
    ret

```

Figure 6: A sample instruction implementation

V. Results

The simulator described above required approximately fifteen Pentium instructions to process each simulated CPU instruction. This includes the instruction specific code plus the constant overhead of the fetch-decode-execute loop that is shown in Figure 5. While the instruction ratio can provide an initial indication of relative times, on modern processors actual time penalties are dependent on the host computer architecture and performance must be verified through benchmarking.

To provide a benchmark performance estimate, programs were developed to compare the execution time for each simulated instruction with its Pentium equivalent over a large number of repetitions. Specifically, the resulting programs were executed on a 1.73 GHz Pentium, 512M notebook under Linux. Each simulated instruction and its equivalent Pentium counterpart was executed ten million times. Table 1 shows the average time for the load, increment, exclusive or, and compare instructions derived from the UNIX “time” command. The times required to execute the programs running under the CPU simulator were between ten and thirty times greater than the times necessary for the equivalent Pentium instructions.

With benchmarks indicating approximately an order-of-magnitude speed penalty, verification was necessary to ensure that the simulated CPU was capable of controlling an actual robot in real time. To this end, robot control routines were developed in the simulated instruction set to move the robot forward, follow the left-hand wall, sense a barrier, and turn right. These routines were combined to form a program that allowed the robot to follow a wall in a simple, continuous loop. These programs were hand-translated into machine code, then placed into the CPU simulator memory for execution. A sample routine written with the simulated instruction set to move the robot and test the sensor values is shown in Figure 7.

Table 1: Instruction timing comparisons

Instruction	Average User Time (seconds)	Ratio
ldi (load immediate)		
cpu simulator	0.080	11.0
intel assembly	0.007	
inc (increment)		
cpu simulator	0.076	13.5
intel assembly	0.006	
eor (exclusive or)		
cpu simulator	0.096	17.9
intel assembly	0.005	
cmp (compare)		
cpu simulator	0.091	27.2
intel assembly	0.003	


```

# initialization
rinit
ldi R1,$256          # R1 = the address for sensor values
ldi R5,$300         # R5 = value for too close to obstacle

# move robot forward
ldi R3,$1           # R3 = left motor speed
ldi R4,$1           # R4 = right motor speed
rspeed R3,R4

# check left sensor
rsens R1
ld R2,memaddr       # R2 = first sensor value returned
cmp R2,R5
bg address

```

Figure 7: A sample routine to move the robot and test a sensor value

The resulting program executed by the CPU simulator was able to control both the simulated Khepera and the actual Khepera II robot. When tested on a Pentium 2.2 GHz, 512M notebook computer running Linux and using a serial port to communicate with the robot, the Khepera II robot was able to respond to commands at a forward speed of up to 8 cm/sec. before the robot speed exceeded the ability to properly react. The program was also able to control the robot while following walls to successfully navigate a rectangular area. This compares favorably with native Pentium code in the same environment, which sustained robot speeds up to 10 cm/sec.

VI. Observations

This study established that it is feasible to control both the robot simulation as well as an actual robot using a CPU simulator designed specifically for this task. The success obtained with a limited machine instruction set and a minimum number of robot control instructions suggested the possibility of increasing the sophistication of the instruction set to give students a more realistic implementation project.

To this end, several changes were implemented. These changes include specifying a variety of memory addressing modes including register, constant, memory address, and indirect register references. The program control instructions were expanded, the compare/branch instructions reworked, and minor organizational changes were incorporated. While under continuous improvement, a typical expanded instruction set is shown in Figure 8. These instructions were not specifically tested for speed since they formed a pedagogical addition to the set already determined to be sufficient for real-time robot control.


```

# load instruction formats:
# (also applies to and, or, eor, and add instructions)
ld    Rd,Rs          # register to register
ld    Rd,$immed     # immediate to register
ld    Rd,memaddr    # memory address to register
ld    Rd,memaddr[Rs] # memory addr + register to register
ld    Rd,*Rs        # register indirect to register

# store instruction formats:
st    Rd,memaddr    # register to memory address
st    Rd,memaddr[Rs] # register to memory addr + register
st    Rd,*Rs        # register to register indirect

# jump instruction formats:
jmp   location      # jump to location
jmp   *Rd           # jump register indirect
jal   Rd,location   # jump and link to location
jgt   Rd,Rs,location # jump to location if Rd > Rs
jgt   Rd,$immed,location # jump to location if Rd > $immed
jlt   Rd,Rs,location # jump to location if Rd < Rs
jlt   Rd,$immed,location # jump to location if Rd < $immed
jeq   Rd,Rs,location # jump to location if Rd = Rs
jeq   Rd,$immed,location # jump to location if Rd = $immed
jez   Rd,location   # jump to location if Rd = 0

# robot instruction formats:
rinit                # initialize communication
rsens                # read robot sensors (into registers)
rspeed Rright,Rleft  # set robot motor speeds
rspeed $immed,$immed # set robot motor speeds

```

Figure 8: Expanded instruction set formats

Finally, to help students create robot control programs in the simulated environment, a simple assembler has been deployed to avoid hand-translating the virtual CPU instructions into machine code, as described in [7].

VII. Summary

This study established that a simulated CPU is capable of controlling both simulated and actual robots. We believe that having students implement their own simulator for an instruction set including register and memory access modes, along with writing subsequent robot-control programs on their own CPU, provides a rich environment for a deeper understanding of the CPU impact on program development.

References

- [1] Wolfer, J. and H. Rababaah. "Creating a Hands-On Environment for Teaching Assembly Language Programming." Global Congress on Engineering: Technology Education, 2005.
- [2] Bem, Ewa Z. and Luke Petelczyc. "MiniMIPS – A Simulation Project for the Computer Architecture Laboratory." Proceedings of the 34th SIGCSE Technical Symposium in Computer Science Education, Reno, Nevada, Feb. 19 -13, 2003.
- [3] Ellard, Daniel et al. "On the Design of a New CPU Architecture for Pedagogical Purposes." Proceedings of the Ninth Workshop on Computer Architecture Education, Anchorage, Alaska, May 2002.
- [4] Scott, Kirk. "MISC: The Minimal Instruction Set Computer." Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 2002.
- [5] K-Team S.A. *Khepera II User Manual*. version 1.1. K-Team, 2002.
- [6] Michael, O. "Khepera Simulator package version 2.0: Freeware mobile robot simulator." written at the University of Nice Sophia-Antipolis by Oliver Michael. Downloadable from the World Wide Web, <http://diwww.epfl.ch/lami/team/michel/khep-sim/index.html>, accessed 3/2008.
- [7] Susan L. Gordon and James Wolfer. "A Python-Based Assembler for a Custom, Robot-Centric, Instruction Set.", Proceedings of the International Conference on Engineering and Computer Education, Santos, Brazil, 2007.

Appendix

```
# Program:    cpu.s
# Function:   CPU simulator for passing robot commands
# Updated:   c335, February 7, 2006 - CPU project
```

```
        .section .data
```

```
=====
#
#   Simulator Memory, 16bit address, 64K max
#   A program to make the robot turn right at a barrier and follow left wall is
#   loaded into the memory for testing.
#=====
#
#   Simulator register use for the test program:
#       r1 = data area pointer
#       r2 = compare register
#   Registers 5 through 8 are used to hold the threshold values
#   that will be compared to the robot sensor data returned by
#   the 'rsens' command:
#       r5 = wall ahead
#       r6 = clear ahead
#       r7 = side too close
#       r8 = side too far
#=====
```

```
smem:
```

```

                                #initialization
        .byte 0x0F,0x00          # rinit
        .byte 0x07,0x01          # ldi r1,(64*4)
        .long 64*4
        .byte 0x07,0x05          # ldi r5,300
        .long 300
        .byte 0x07,0x06          # ldi r6,60
        .long 60
        .byte 0x07,0x07          # ldi r7,300
        .long 300
        .byte 0x07,0x08          # ldi r8,50
        .long 50

                                #go_forward
        .byte 0x07,0x03,0x01,0x00,0x00,0x00    # ldi r3,1
        .byte 0x07,0x04,0x01,0x00,0x00,0x00    # ldi r4,1
        .byte 0x11,0x34                      # rspeed r3,r4

                                #check_forward_sensors
        .byte 0x10,0x01                      # rsens r1
        .byte 0x08,0x02,0x0C,0x01,0x00,0x00    # ld r2,(rsloc+12)
```

```

.byte 0x0A,0x25          # cmp r2,r5
.byte 0x0B,0x00,0x6A,0x00,0x00,0x00 # bg (look_left)
                                #turn_hard_right
.byte 0x07,0x03,0x01,0x00,0x00,0x00 # ldi r3,1
.byte 0x07,0x04,0xFF,0xFF,0xFF,0xFF # ldi r4,-1
.byte 0x11,0x34          # rspeed r3,r4
                                #check_clear
.byte 0x10,0x01          # rsens r1
.byte 0x08,0x02,0x04,0x01,0x00,0x00 # ld r2,(rsloc+4)
.byte 0x0A,0x26          # cmp r2,r6
.byte 0x0C,0x00,0x4C,0x00,0x00,0x00 # bl (check_clear)
                                #reset_forward
.byte 0x07,0x03,0x01,0x00,0x00,0x00 # ldi r3,1
.byte 0x07,0x04,0x01,0x00,0x00,0x00 # ldi r4,1
.byte 0x11,0x34          # rspeed r3,r4
                                #look_left
.byte 0x10,0x01          # rsens r1
.byte 0x08,0x02,0x00,0x01,0x00,0x00 # ld r2,(rsloc)
                                #too_close
.byte 0x0A,0x27          # cmp r2,r7
.byte 0x0B,0x00,0x8E,0x00,0x00,0x00 # bg (too_far)
                                #go_right
.byte 0x07,0x03,0x02,0x00,0x00,0x00 # ldi r3,2
.byte 0x07,0x04,0x01,0x00,0x00,0x00 # ldi r4,1
.byte 0x11,0x34          # rspeed r3,r4
.byte 0x0E,0x00,0x20,0x00,0x00,0x00 # jmp (go_forward)
                                #too_far
.byte 0x0A,0x28          # cmp r2,r8
.byte 0x0C,0x00,0x20,0x00,0x00,0x00 # bl (go_forward)
                                #go_left
.byte 0x07,0x03,0x01,0x00,0x00,0x00 # ldi r3,1
.byte 0x07,0x04,0x02,0x00,0x00,0x00 # ldi r4,2
.byte 0x11,0x34          # rspeed r3,r4
.byte 0x0E,0x00,0x20,0x00,0x00,0x00 # jmp (go_forward)

.rept 65536-170
.byte 0xff
.endr

```

```

#=====
# Simulator registers, 16 32-bit registers
# Initialized to 0x00 - 0x0F on "boot"
#=====

```

regs:

```
.long 0,1,2,3,4,5,6,7
```

```
.long 8,9,10,11,12,13,14,15
# These names were used for debugging:
rsloc = smem+(64*4) # rsens output memory address
reg0 = regs          # reg0 will be flag register for compare
reg1 = regs + 4      # reg1 will be address for robot output
reg2 = regs + 8      #
reg3 = regs + 12
reg4 = regs + 16
reg5 = regs + 20
reg6 = regs + 24
reg7 = regs + 28
reg8 = regs + 32
reg9 = regs + 36
rega = regs + 40
regb = regs + 44
regc = regs + 48
regd = regs + 52
rege = regs + 56
regf = regs + 60
```

```
#=====
#   Operation dispatch table
#   It contains the routine to process each op-code, 256 max
#=====
```

```
optbl:
.long  mov00
.long  and01
.long  or02
.long  eor03
.long  not04
.long  inc05
.long  dec06
.long  ldi07
.long  ld08
.long  st09
.long  cmp0A
.long  bg0B
.long  bl0C
.long  beq0D
.long  jmp0E
.long  rinit
.long  rsens
.long  rspeed
.rept  256-18
.long  opFF
.endr
```

```

#=====
#   Misc. data follows
#=====
# Control String for op-codes not yet implemented
nostr: .string "Not Implemented: %d\n"

# Robot Sensor Control String
rsctl: .string "N\n"

# Robot Command to Set Speed
rspeedctl:
    .ascii "D,%d,%d\n\0" # Sets left and right motor speeds
rspeedbuf:
    .space 80          # buffer to hold command formatted by sprintf

# Temporary storage for simulated IP
ipsv: .long 0

# Extra control strings used for debugging:
# Control strings for register print
reg_print1:
    .string "Reg0 - %x, Reg1 - %x, Reg2 - %x, Reg3 - %x\n"
reg_print2:
    .string "Reg4 - %x, Reg5 - %x, Reg6 - %x, Reg7 - %x\n"
reg_print3:
    .string "Reg8 - %x, Reg9 - %x, RegA - %x, RegB - %x\n"
reg_print4:
    .string "RegC - %x, RegD - %x, RegE - %x, RegF - %x\n"
# Control string for robot sensors print
rsprt: .string "Robot Sensors:      %d,%d,%d,%d,%d,%d,%d,%d\n"

    .globl _start
    .section .text

#####
#
#   Simulator CPU: Fetch, Decode, Execute
#
#   - Register %edi is used as the simulated instruction pointer
#   - Register %eax is a scratch register
#   - The simulated 64K memory is smem
#   - 'optbl' is the instruction dispatch table index by operation
#

```

```
*****
```

```
_start:
    movl  $0,%edi          # simulated ip
fetch:
    andl  $0xffff,%edi     # Mod 65536
    movb  smem(,%edi,1),%al # Fetch opcode
    andl  $0xff,%eax       # isolate opcode in register
    movl  optbl(,%eax,4),%eax # Decode: get address of routine
    incl  %edi             # Bump ip to next byte of operation
    call  *%eax            # Execute the instruction
#    call  regprint        # For debugging only
    jmp   fetch
```

```
#=====
#   Instruction Execution Routines
#=====
#-----
#   Move source register,Rs, to destination register, Rd.
#   format - mov  Rd,Rs
```

```
-----
mov00:
    movb  smem(,%edi,1),%al
    andl  $0xff,%eax
    movl  %eax,%ebx
    andl  $0x0f,%eax       # Rs number in %eax
    sarl  $4,%ebx          # Rd number in %ebx
    movl  regs(,%eax,4),%eax # Rs value to %eax
    movl  %eax,regs(,%ebx,4) # Value to Rd
    incl  %edi             # Point ip to next instruction
    ret
```

```
#-----
#   Bitwise logical and of Rs with Rd with the result in Rd.
#   format - and  Rd,Rs
```

```
-----
and01:
    movb  smem(,%edi,1),%al
    andl  $0xff,%eax
    movl  %eax,%ebx
    andl  $0x0f,%eax       # Rs number in %eax
    sarl  $4,%ebx          # Rd number in %ebx
    movl  regs(,%eax,4),%eax # Rs value to %eax
    andl  regs(,%ebx,4),%eax # and with Rd value
    movl  %eax,regs(,%ebx,4) # Result to Rd
    incl  %edi             # Point ip to next instruction
```



```

ret
#-----
#   Bitwise logical or of Rs with Rd with the result in Rd
#   format - or Rd,Rs
#-----
or02:
    movb smem(,%edi,1),%al
    andl $0xff,%eax
    movl %eax,%ebx
    andl $0x0f,%eax      # Rs number in %eax
    sarl $4,%ebx        # Rd number in %ebx
    movl regs(,%eax,4),%eax # Rs value to %eax
    orl  regs(,%ebx,4),%eax # or with Rd value
    movl %eax,regs(,%ebx,4) # Result to Rd
    incl %edi           # Point ip to next instruction
    ret

#-----
#   Bitwise logical eor of Rs with Rd with the result in Rd.
#   format - eor Rd,Rs
#-----
eor03:
    movb smem(,%edi,1),%al
    andl $0xff,%eax
    movl %eax,%ebx
    andl $0x0f,%eax      # Rs number in %eax
    sarl $4,%ebx        # Rd number in %ebx
    movl regs(,%eax,4),%eax # Rs value to %eax
    xorl regs(,%ebx,4),%eax # eor with Rd value
    movl %eax,regs(,%ebx,4) # Result to Rd
    incl %edi           # Point ip to next instruction
    ret

#-----
#   Bitwise logical not of Rd with the result in Rd
#   format - not Rd
#-----
not04:
    movb smem(,%edi,1),%al
    andl $0x0f,%eax      # Rd number in %eax
    notl regs(,%eax,4)    # not Rd value
    incl %edi           # Point ip to next instruction
    ret

```

```
#-----
#   Add 1 to Rd.
#   format - inc Rd
#-----
```

```
inc05:
    movb smem(,%edi,1),%al
    andl $0x0f,%eax      # Rd number in %eax
    incl regs(,%eax,4)   # Increase Rd value
    incl %edi            # Point ip to next instruction
    ret
```

```
#-----
#   Subtract 1 from Rd.
#   format - dec Rd
#-----
```

```
dec06:
    movb smem(,%edi,1),%al
    andl $0x0f,%eax      # Rd number in %eax
    decl regs(,%eax,4)   # Decrease Rd value
    incl %edi            # Point ip to next instruction
    ret
```

```
#-----
#   Load an immediate value into register Rd.
#   format - ldi Rd,$immediate
#-----
```

```
ldi07:
    movb smem(,%edi,1),%al
    andl $0x0f,%eax      # Rd number in %eax
    incl %edi            # Next location in storage is the
    movl smem(,%edi,1),%ebx # 4 byte immediate value to %ebx
    movl %ebx,regs(,%eax,4) # Move the value to specified register
    addl $4,%edi         # Point ip to next instruction
    ret
```

```
#-----
#   Load a value from storage into register Rd.
#   format - ld Rd,address
#-----
```

```
ld08:
    movb smem(,%edi,1),%al
    andl $0x0f,%eax      # Rd number in %eax
    incl %edi            # Next location in storage is the
    movl smem(,%edi,1),%ebx # address of data to move to %ebx
    movl smem(,%ebx,1),%ebx
```

```

    movl  %ebx,regs(,%eax,4) # Move the value to specified register
    addl  $4,%edi           # Point ip to next instruction
    ret

#-----
#   Put a value from a register into storage at (smem + address).
#   format - st Rd,address
#-----
st09:
    movb  smem(,%edi,1),%al
    andl  $0x0f,%eax       # Rs number in %eax
    movl  regs(,%eax,4),%eax # Value in register to %eax
    incl  %edi             # Next location in storage is the
    movl  smem(,%edi,1),%ebx # offset in memory for value
    movl  %eax,smem(,%ebx,1) # Value from Rs to memory
    addl  $4,%edi           # Point ip to next instruction
    ret

#-----
#   Compare value in R2 to R1 with the result code in reg 0 - 00 = less
#                                           01 = greater
#                                           02 = equal
#   format - cmp R1,R2
#-----
cmp0A:
    movb  smem(,%edi,1),%al
    andl  $0xff,%eax
    movl  %eax,%ebx
    andl  $0x0f,%eax       # R2 number in %eax
    sarl  $4,%ebx          # R1 number in %ebx
    movl  $0,regs          # Clear reg0 for flag values
    movl  regs(,%eax,4),%eax # Value in register to %eax
    movl  regs(,%ebx,4),%ebx # Value in register to %ebx
    cmpl  %ebx,%eax        # Compare R2,R1
    jle   less_or_equal
    movl  $1,reg0          # Code for greater = 1
    jmp   end_cmp
less_or_equal:
    jl    end_cmp          # Code for less = 0
    movl  $2,reg0          # Code for equal = 2
end_cmp:
    incl  %edi             # Point ip to next instruction
    ret

```

```
#-----  
#   Branch greater - used with compare R2 to R1 instruction tests result code in reg0,  
#   01 = greater.  
#   format - bg address  
#-----  
bg0B:  
    incl    %edi                # Move to address part of instruction  
    cmpl   $1,reg0            # Code for greater  
    jne    no_bg  
    movl   smem(%edi,1),%edi  # Move specified address to ip  
    jmp    fetch  
no_bg:  
    addl   $4,%edi            # Move ip past branch address not taken  
    ret  
  
#-----  
#   Branch less - used with compare R2 to R1 instruction tests result code in reg0,  
#   00 = less.  
#   format - bl address  
#-----  
bl0C:  
    incl    %edi                # Move to address part of instruction  
    cmpl   $0,reg0            # Code for less  
    jne    no_bl  
    movl   smem(%edi,1),%edi  # Move specified address to ip  
    jmp    fetch  
no_bl:  
    addl   $4,%edi            # Move ip past branch address not taken  
    ret  
  
#-----  
#   Branch equal - used with compare R2 to R1 instruction tests result code in reg0,  
#   02 = equal.  
#   format - beq address  
#-----  
beq0D:  
    incl    %edi                # Move to address part of instruction  
    cmpl   $2,regs            # Code for equal  
    jne    no_beq  
    movl   smem(%edi,1),%edi  # Move specified address to ip  
    jmp    fetch  
no_beq:  
    addl   $4,%edi            # Move ip past branch address not taken  
    ret
```

```

#-----
#   Unconditional branch.
#   format - jmp address
#-----
jmp0E:
    incl    %edi
    movl   smem(,%edi,1),%edi # Move specified address to ip
    jmp    fetch

#-----
#   rinit -- Open pipes to robot.
#-----
rinit:
    call   open_pipes
    incl   %edi                # Point ip to next instruction
    ret

#-----
#   rsens -- Read robot sensors.
#-----
rsens:
    movb   smem(,%edi,1),%al
    andl   $0xFF,%eax
    movl   regs(,%eax,4),%eax # Rd value to %eax = offset
    addl   $smem,%eax        # Get storage addr in smem
    pushl  $rsctl            # Push command string
    pushl  %eax              # Push addr to return data
    movl   %edi,ipsv         # Save ip
    call   sndRcv_0          # Issue robot command
    addl   $8,%esp

#   pushl  rsloc+28          # Call to print sensors
#   pushl  rsloc+24          #   just for debugging
#   pushl  rsloc+20
#   pushl  rsloc+16
#   pushl  rsloc+12
#   pushl  rsloc+8
#   pushl  rsloc+4
#   pushl  rsloc
#   pushl  $rsprt
#   call   printf
#   addl   $36,%esp

    movl   ipsv,%edi         # Restore ip
    incl   %edi              # Point ip to next instruction

```

```

ret
#-----
#   Set robot speed.
#   format - rspeed Rleft, Rright
#           Rleft - register containing left motor speed
#           Rright - register containing right motor speed
#           valid speed values are -127 to 127
#           reg1 has response address for testing
#-----
rspeed:
    movl   %edi,ipsv          # Save ip register before call
    movb   smem(,%edi,1),%al
    andl   $0xff,%eax
    movl   %eax,%ebx
    andl   $0x0f,%eax        # Rr to %eax
    movl   regs(,%eax,4),%eax # Move right speed value to %eax
    sarl   $4,%ebx          # Rl to %ebx
    movl   regs(,%ebx,4),%ebx # Move left speed value to %ebx
    pushl  %eax
    pushl  %ebx
    pushl  $rspeedctl
    pushl  $rspeedbuf
    call   sprintf
    pushl  $rsloc
    call   sndRcv_0

    addl   $20,%esp

    movl   ipsv,%edi
    incl   %edi              # Point ip to next instruction
    ret

#-----
#   OpFF -- Temporary operation
#-----
opFF:
    pushl  $0xFF #Print its own op-code
    jmp    notimp

#-----
#   Notimp: Default operation for operations not implemented
#   Should never happen in actual program, represents an
#   error. Treat as no-op for now.
#-----
notimp:
    movl   %edi,ipsv          # In case printf changes ip

```

```
    pushl $nostr          # Pointer to control string
    call  printf
    addl  $8,%esp
    movl  ipsv,%edi      # Restore simulator ip
    call  exit           # Stop program on bad op
    ret                # Should never get here, but in case

#-----
#    regprint - print registers for debugging
#-----
regprint:
    movl  %edi,ipsv      # In case printf changes ip
    pushl reg3          # Register values
    pushl reg2
    pushl reg1
    pushl reg0
    pushl $reg_print1   # Pointer to control string
    call  printf
    addl  $20,%esp
    pushl reg7          # Register values
    pushl reg6
    pushl reg5
    pushl reg4
    push  $reg_print2   # Pointer to control string
    call  printf
    addl  $20,%esp
    pushl regb          # Register values
    pushl rega
    pushl reg9
    pushl reg8
    push  $reg_print3   # Pointer to control string
    call  printf
    addl  $20,%esp
    pushl regf          # Register values
    pushl rege
    pushl regd
    pushl regc
    push  $reg_print4   # Pointer to control string
    call  printf
    addl  $20,%esp
    movl  ipsv,%edi      # Restore simulator ip
    ret
```