

A Web-Based Graphical Education Support Tool for Teaching the Course of Assembly Language Programming

URL: <http://cis.k.hosei.ac.jp/~yamin/asm/>

Yamin Li, Department of Computer Science, Hosei University, yamin@hosei.ac.jp

Assembly Language Programming

- The foundation of many machines, from general-purpose computer systems to embedded systems, lies in assembly language programming and computer architecture
- If someone doesn't understand assembly language, he (she) can neither develop a compiler nor design a CPU
- The assembly language can be used to demonstrate what is actually happening inside the computer
- It is an important path to understanding how the computer works at the machine level

Assembly Language Programming

- Some special (hardware) operations may not be controlled in C or Java
- Example: How to select one of the following round-modes of FPU in C?
 - Round to nearest (to even if g r s = 1 0 0)
 - Round toward minus infinity ($-\infty$)
 - Round toward plus infinity ($+\infty$)
 - Round toward zero (also called truncate)
- Some assembly codes must be inserted in C

```
#define Near asm volatile("fldcw _RoundNear")  
int _RoundNear = 0x103f; // Round mode = 00  
Near; // Round to nearest
```

Top Three Assembly Programming Languages

- High-level languages are machine independent
- Assembly languages are machine dependent
- Top three assembly programming languages
 - Intel x86
 - ARM
 - MIPS
- We selected MIPS instruction set
 - MIPS is a typical RISC instruction set
 - To design MIPS CPUs in the following course of **Computer Organization and Design**

Functions of Simulators for Assembly Programming

- The basic assembly programs' **assembling** and simulation:
 - Grammar checking
 - Translating to binary code
 - Single-step execution
 - Break-point setting
 - Showing the contents of the register file
 - Showing the contents of the data memory
- The basic input and output system function calls, like **scanf()** and **printf()** in C, or **readLine()** and **System.out.print()** in Java

Two Existing MIPS Simulators

- SPIM — Windows (PCSpim) and Linux (spim and xspim)
 - Register display that shows the values of registers
 - Control buttons that let us interact with simulator
 - Text segment that display instructions
 - Data and stack segment
 - Spim messages used to write messages
- MARS — Java-based simulator
 - Provided a self-contained program editor
 - Showed the displacements with labels
 - Changed the simulator layout

AsmSim — A MIPS Simulator We Developed

- Web-based — does not need installation
- Graphical — supports VGA
- The AsmSim provides three windows:
 - A main window that provides control buttons and displays the instructions, the values of registers, and the data of the memory
 - A graphics console window that is used to input command, to display messages, and to show the images of Video RAM (VRAM)
 - An online editor window for editing the assembly program

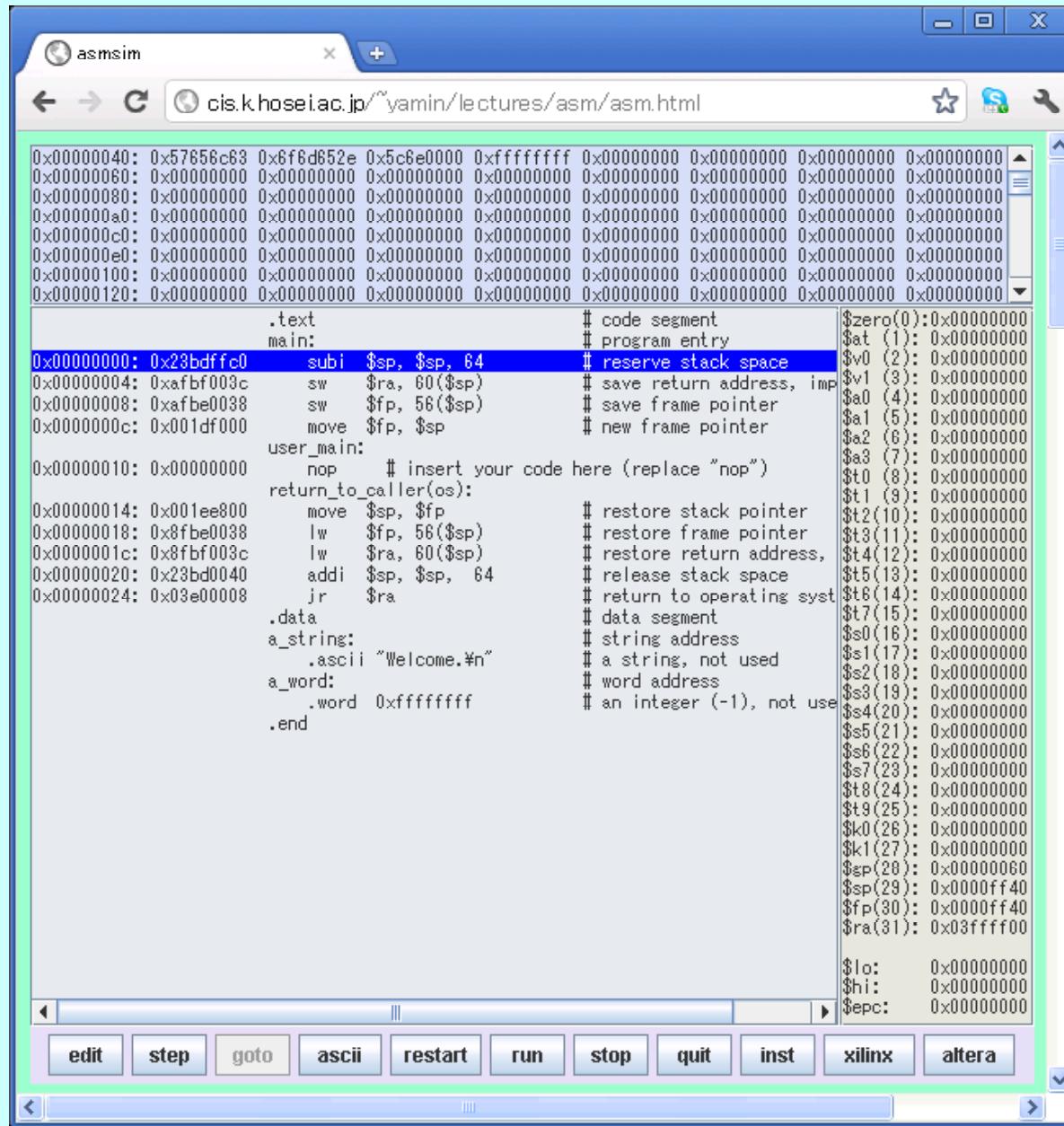
An Assembly Program Example

```
/*
 * hello.s
 * prints "Hello, World!" to console window
 * input: $4: the string address
 */
.text
main:
    subi    $sp, $sp, 24      # code segment
    sw      $ra, 20($sp)       # program entry
    sw      $fp, 16($sp)       # reserve stack space
    move   $fp, $sp           # save return address
    move   $fp, $sp           # save frame pointer
    move   $fp, $sp           # new frame pointer
user_main:
    la     $4, msg            # hello message address
    jal    printf             # print out message
return_to_caller:
    move  $sp, $fp            # exit()
    move  $fp, 16($sp)         # restore stack pointer
    lw    $fp, 16($sp)         # restore frame pointer
    lw    $ra, 20($sp)         # restore return address
    addu $sp, $sp, 24          # release stack space
    jr    $ra                 # return to operating system
.data
msg:
    .ascii "Hello, World!\n"  # the string
.end
```

The Functions of AsmSim

- The basic assembling and simulation
- The basic input and output system function calls
- Interrupt mechanism for handling the keyboard interrupt
- Manipulating a standard VGA (640×480 pixels)
- Graphics object draw and fill
- Generating Xilinx COE and Altera MIF for FPGAs
- Some other useful function calls, like `get a random number`, `get the timer`, `sleep`, and `get calendar`
- Supporting the data structure of the `linked-list` for implementing some graph algorithms

The Main Window of AsmSim



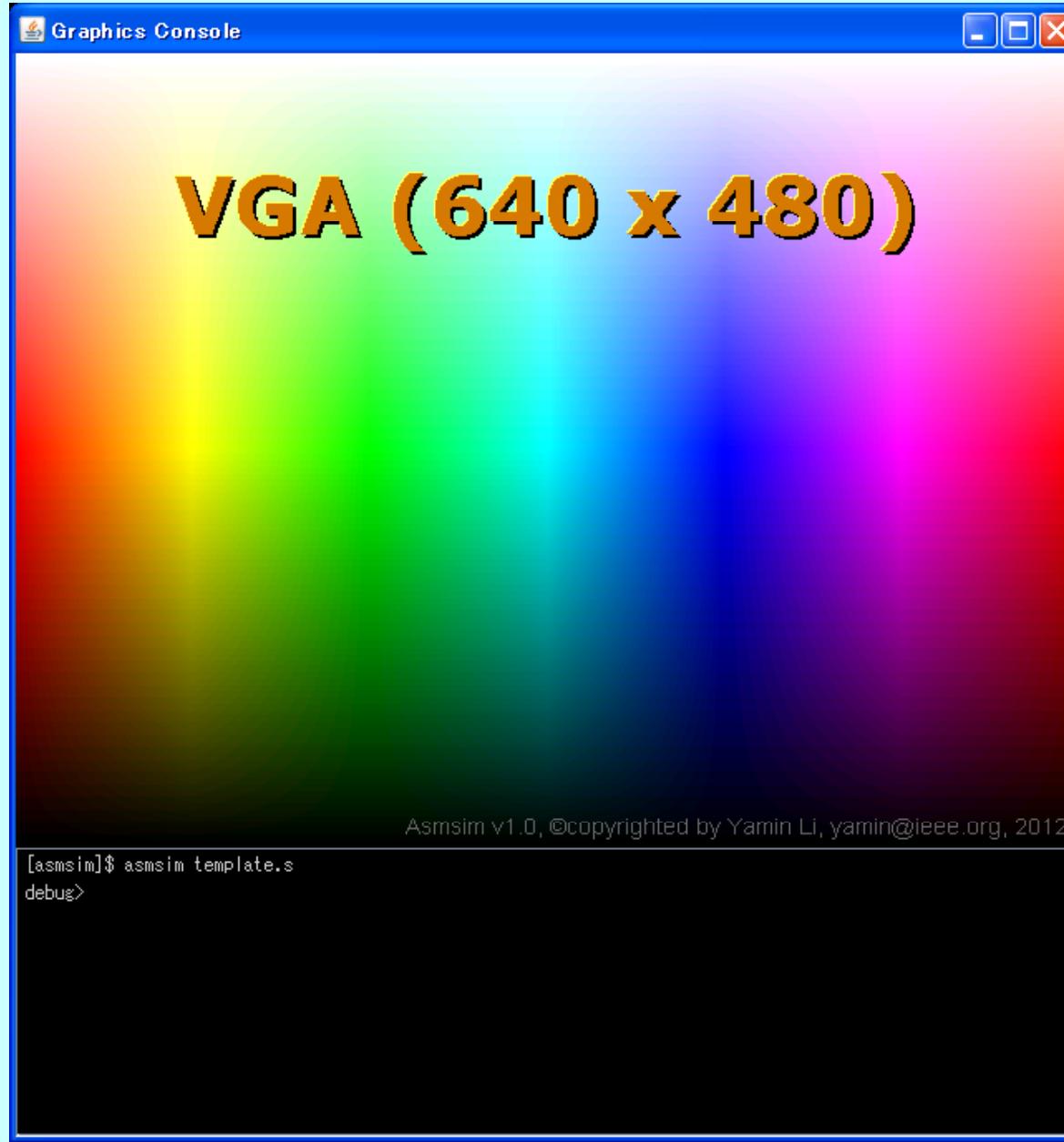
The Highlighted Program Edit Window of AsmSim

The screenshot shows the 'Program Editor' window of AsmSim. The window title is 'Program Editor'. The code editor displays assembly code for a C program template. The code includes comments explaining the purpose of each assembly instruction. The assembly code is as follows:

```
1 /*  
2 * template.s, for editing your codes  
3 * Control_s for searching (quit search with mouse or arrow key pressing)  
4 * Copy & Paste: See http://cis.k.hosei.ac.jp/~yamin/asm/CopyAndPaste.html  
5 */  
6 .text  
7 main:  
8     subu $sp, $sp, 64          # code segment  
9     sw    $ra, 60($sp)         # program entry  
10    sw    $fp, 56($sp)         # reserve stack space  
11    move  $fp, $sp             # save return address, important  
12    user_main:  
13        nop      # insert your code here (replace "nop")  
14    return_to_caller(os):  
15        move   $sp, $fp           # save frame pointer  
16        lw     $fp, 56($sp)         # restore stack pointer  
17        lw     $ra, 60($sp)         # restore frame pointer  
18        addu  $sp, $sp, 64           # restore return address, important  
19        jr     $ra               # release stack space  
20    .data  
21    a_string:                 # return to operating system  
22        .ascii "Welcome.\n"  
23    a_word:                   # data segment  
24        .word  0xffffffff          # string address  
25    .end  
26  
27  
28  
29  
30  
31  
32  
33  
34
```

At the bottom of the window, there is a button labeled 'Assemble'.

The Graphics Console Window of AsmSim



11 Buttons in the Main Window

1. **(edit)**: opens the program edit window
2. **(step)**: executes one instruction that is highlighted
3. **(goto)**: executes instructions to the break-point
4. **(ascii)**: displays the ascii of selected data
5. **(restart)**: re-loads the user program
6. **(run)**: executes user program
7. **(stop)**: stops the execution
8. **(quit)**: quits and enters *command line mode*
9. **(inst)**: displays the encodes of the MIPS instructions
10. **(xilinx)**: generates Xilinx COE file
11. **(altera)**: generates Altera MIF file

Instructions AsmSim Supported

MIPS32 Instruction Format (32 bits)						rs, rt, rd, sa: 5-bit number;	imm: 16-bit number;	addr: 26-bit number
op	rs	rt	rd	sa	func	Instructions	Operations	
000000	rs	rt	rd	00000	100000	add \$rd, \$rs, \$rt	reg[rd] <- reg[rs] + reg[rt], reg[rd] <- reg[rs] + reg[rt], reg[rd] <- reg[rs] - reg[rt], reg[rd] <- reg[rs] & reg[rt], reg[rd] <- reg[rs] reg[rt], reg[rd] <- reg[rs] < reg[rt],	pc <- pc + 4 (addu: unsigned add) pc <- pc + 4 (subu: unsigned sub) pc <- pc + 4 pc <- pc + 4 pc <- pc + 4 pc <- pc + 4 (true: 1; false: 0) pc <- pc + 4
000000	rs	rt	rd	00000	100001	addu \$rd, \$rs, \$rt	reg[rd] <- reg[rs] + reg[rt],	pc <- pc + 4
000000	rs	rt	rd	00000	100010	sub \$rd, \$rs, \$rt	reg[rd] <- reg[rs] - reg[rt],	pc <- pc + 4
000000	rs	rt	rd	00000	100011	subu \$rd, \$rs, \$rt	reg[rd] <- reg[rs] - reg[rt],	(subu: unsigned sub) pc <- pc + 4
000000	rs	rt	rd	00000	100100	and \$rd, \$rs, \$rt	reg[rd] <- reg[rs] & reg[rt],	pc <- pc + 4
000000	rs	rt	rd	00000	100101	or \$rd, \$rs, \$rt	reg[rd] <- reg[rs] reg[rt],	pc <- pc + 4
000000	rs	rt	rd	00000	100110	xor \$rd, \$rs, \$rt	reg[rd] <- reg[rs] ^ reg[rt],	pc <- pc + 4
000000	rs	rt	rd	00000	100110	slt \$rd, \$rs, \$rt	reg[rd] <- (reg[rs] < reg[rt]),	(true: 1; false: 0) pc <- pc + 4
000000	rs	rt	rd	00000	000100	slv \$rd, \$rt, \$rs	reg[rd] <- reg[rt] <> reg[rs],	pc <- pc + 4
000000	rs	rt	rd	00000	000110	sriv \$rd, \$rt, \$rs	reg[rd] <- reg[rt] >> reg[rs],	(logical, unsigned) pc <- pc + 4
000000	rs	rt	rd	00000	000111	srav \$rd, \$rt, \$rs	reg[rd] <- reg[rt] >>> reg[rs],	(arithmetic, signed) pc <- pc + 4
000000	00000	rt	rd	sa	000000	sll \$rd, \$rt, sa	reg[rd] <- reg[rt] <> sa,	pc <- pc + 4
000000	00000	rt	rd	sa	000010	srl \$rd, \$rt, sa	reg[rd] <- reg[rt] >> sa,	(logical, unsigned) pc <- pc + 4
000000	00000	rt	rd	sa	000011	sra \$rd, \$rt, sa	reg[rd] <- reg[rt] >>> sa,	(arithmetic, signed) pc <- pc + 4
000000	rs	00000	00000	00000	001000	jr \$rs	pc <- reg[rs]	
000000	00000	00000	00000	00000	001100	syscall	system call, transfer control to the OS (user programs should use jal)	
000000	rs	rt	00000	00000	011000	mult \$rs, \$rt	(hi,lo) <- reg[rs] * reg[rt], (hi,lo) <- reg[rs] * reg[rt],	(signed) pc <- pc + 4 (unsigned) pc <- pc + 4
000000	rs	rt	00000	00000	011010	div \$rs, \$rt	(hi,lo) <- reg[rs] / reg[rt], (hi,lo) <- reg[rs] / reg[rt],	(hi: remainder, lo: quotient) pc <- pc + 4 (hi: remainder, lo: quotient) pc <- pc + 4
000000	00000	00000	00000	00000	011011	divu \$rs, \$rt	(hi,lo) <- reg[rs] / reg[rt], (hi,lo) <- reg[rs] / reg[rt],	(hi: remainder, lo: quotient) pc <- pc + 4
000000	00000	00000	00000	rd	000000	mfhi \$rd	reg[rd] <- hi,	pc <- pc + 4
000000	00000	00000	00000	rd	000000	mflo \$rd	reg[rd] <- lo,	pc <- pc + 4
000000	rs	00000	00000	00000	010001	mtih \$rs	hi <- reg[rs],	pc <- pc + 4
000000	rs	00000	00000	00000	010011	mtlo \$rs	lo <- reg[rs],	pc <- pc + 4
000111	rs	rt	rd	00000	000010	mul \$rd, \$rs, \$rt	reg[rd] <- reg[rs] * reg[rt], reg[rd] <- reg[rs] * reg[rt],	(low32(signed * signed)) pc <- pc + 4 return from exception (the return address is in \$epc register) pc <- epc
010000	10000	00000	00000	00000	001100	eret	return from exception (the return address is in \$epc register) pc <- epc	
op	rs	rt	rd	imm		Instructions	Operations	
001000	rs	rt	imm			addi \$rt, \$rs, imm	reg[rt] <- reg[rs] + (signExt)imm,	pc <- pc + 4
001100	rs	rt	imm			andi \$rt, \$rs, imm	reg[rt] <- reg[rs] & (zeroExt)imm,	pc <- pc + 4
001101	rs	rt	imm			ori \$rt, \$rs, imm	reg[rt] <- reg[rs] (zeroExt)imm,	pc <- pc + 4
001110	rs	rt	imm			xori \$rt, \$rs, imm	reg[rt] <- reg[rs] ^ (zeroExt)imm,	pc <- pc + 4
001010	rs	rt	imm			slti \$rt, \$rs, imm	reg[rt] <- (reg[rs] < (signExt)imm), (true: 1; false: 0) pc <- pc + 4	
100011	rs	rt	imm			lw \$rt, imm(\$rs)	reg[rt] <- memory[reg[rs] + (signExt)imm],	pc <- pc + 4
101011	rs	rt	imm			sw \$rt, imm(\$rs)	memory[reg[rs] + (signExt)imm] <- reg[rt],	pc <- pc + 4
000100	rs	rt	imm			beq \$rs, \$rt, label	if (reg[rs] == reg[rt]) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
000101	rs	rt	imm			bne \$rs, \$rt, label	if (reg[rs] != reg[rt]) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
000001	rs	00000	imm			bltz \$rs, label	if (reg[rs] < 0) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
000001	rs	00001	imm			bgez \$rs, label	if (reg[rs] >= 0) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
000110	rs	00000	imm			blez \$rs, label	if (reg[rs] <= 0) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
000111	rs	00000	imm			bgtz \$rs, label	if (reg[rs] > 0) pc <- pc + 4 + (signExt)imm << 2; else pc <- pc + 4	
001111	00000	rt	imm			lui \$rt, imm	reg[rt] <- imm << 16,	pc <- pc + 4
op	address					Instructions	Operations	
000010	addr			i	label		pc <- (pc + 4)[31:28].addr << 2	
000011	addr			jal	label		pc <- (pc + 4)[31:28].addr << 2, reg[31] <- pc + 4	(should be pc + 8)
001111	00000	rt	imm			lahi \$rt, label	reg[rt] <- label & 0xffff0000,	(lui \$rt, imm) pc <- pc + 4
001101	rt	rt	imm			lalo \$rt, label	reg[rt] <- reg[rt] label & 0x0000ffff, (ori \$rt, \$rt, imm) pc <- pc + 4	
001110	00000	rt	imm			lhi \$rt, imm	reg[rt] <- imm & 0xffff0000,	(lui \$rt, imm) pc <- pc + 4
001101	rt	rt	imm			lilo \$rt, imm	reg[rt] <- reg[rt] imm & 0x0000ffff, (ori \$rt, \$rt, imm) pc <- pc + 4	
001000	rs	rt	imm			subi \$rt, \$rs, imm	reg[rt] <- reg[rs] - (signExt)imm, (addi \$rt, \$rt,-imm) pc <- pc + 4	
000000	rt	rd	00000	000000		move \$rd, \$rt	reg[rd] <- reg[rt],	(sll \$rd, \$rt, 0) pc <- pc + 4
000000	00000	00000	00000	000000		nop	no operation	(sll \$0, \$0, 0) pc <- pc + 4
						li \$rt, imm	lihi + lilo	
						la \$rt, label	lahi + lalo	

Java Applet Window

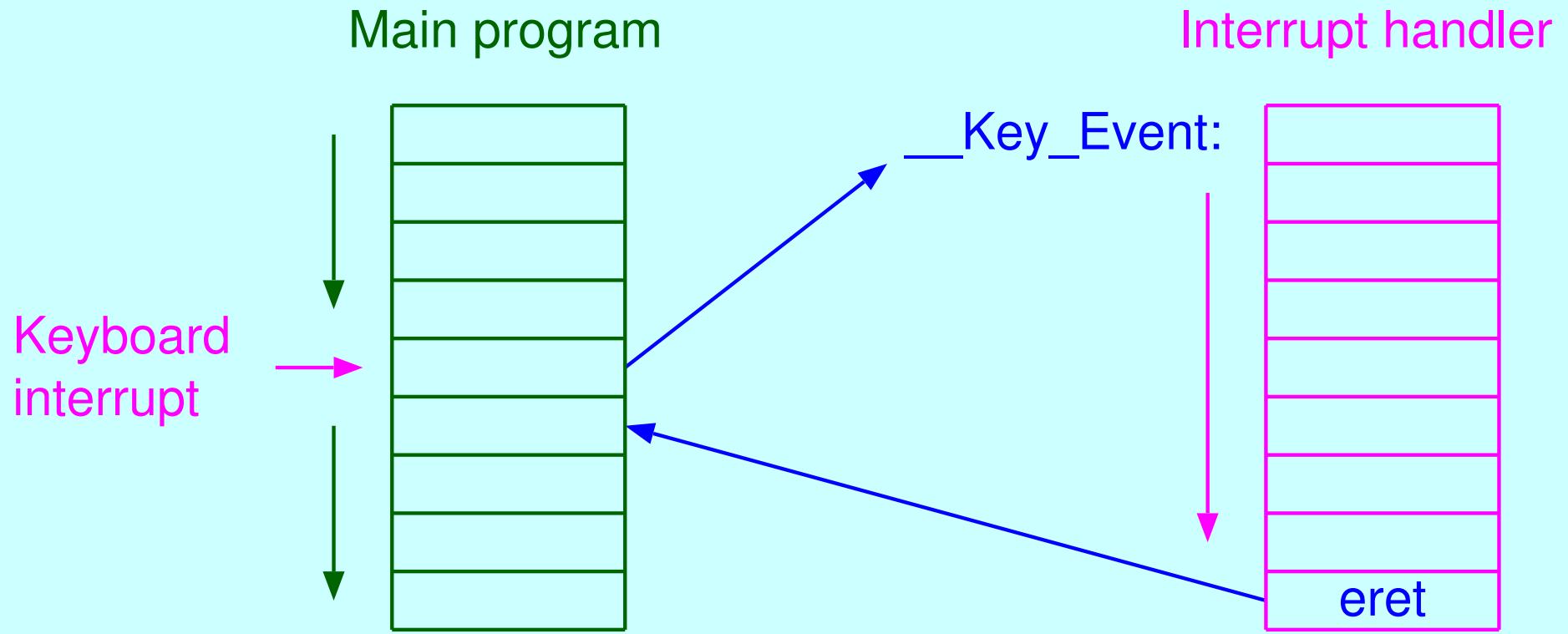
System Calls AsmSim Supported

1. **getchar()** gets a character from STDIN
2. **putchar()** puts a character to STDOUT
3. **scanf()** reads an input from STDIN
4. **printf()** writes output(s) to STDOUT
5. **sprintf()** writes output(s) to a buffer
6. **malloc()** allocates a block of memory
7. **gettimer()** gets the system timer in ms
8. **getrandom()** gets a random integer number
9. **getarrow()** gets an arrow key's information
10. **getcal()** gets the calendar in integers
11. **getcals()** gets the calendar in string
12. **sleep()** suspends execution for an interval of time
13. **paint()** shows the image on the VGA

System Calls AsmSim Supported

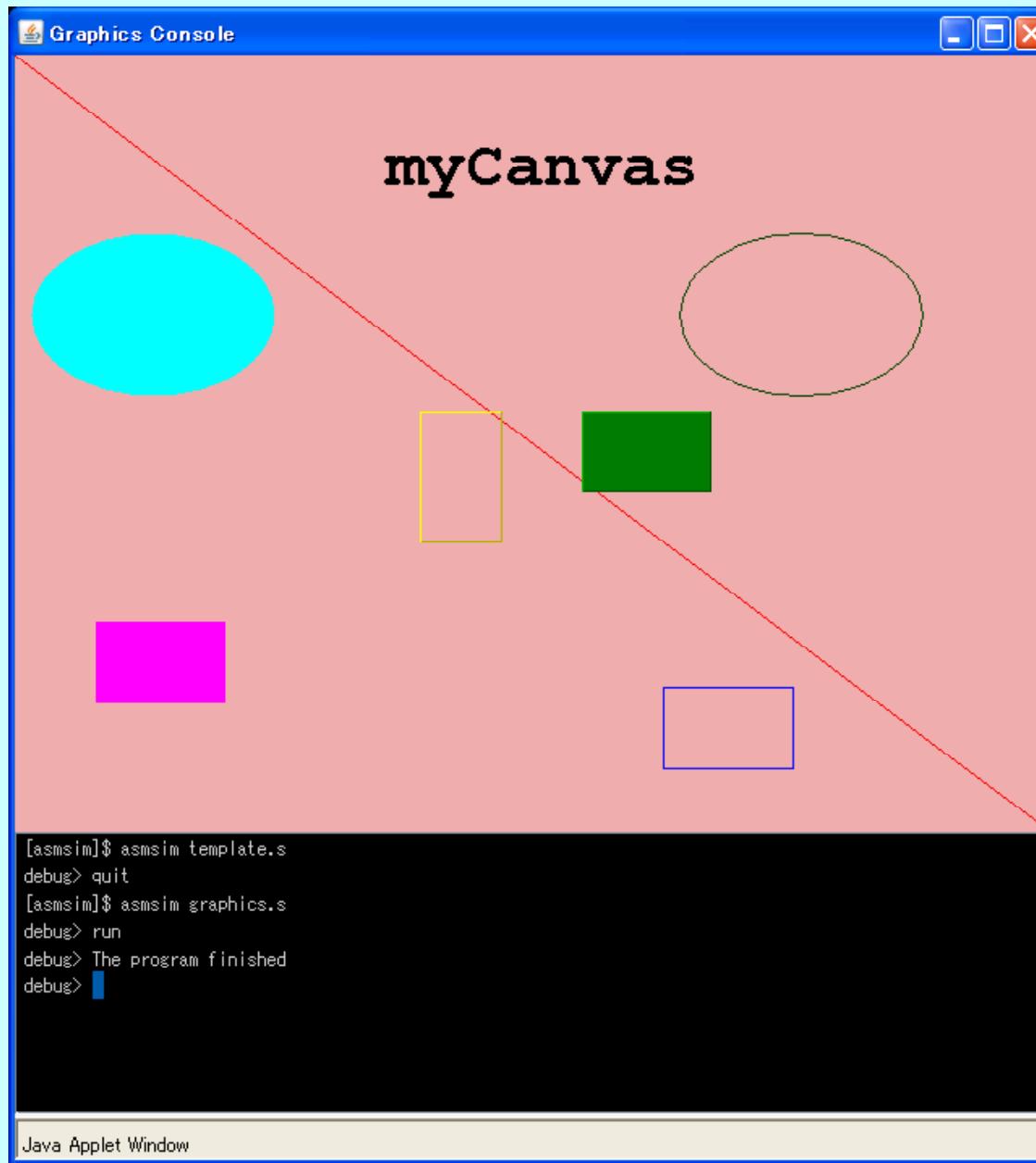
14. **drawline()** draws a color line
15. **drawoval()** draws a color oval
16. **drawrect()** draws a color rectangle
17. **drawrect3d()** draws a color 3D rectangle
18. **drawstring()** draws specified text
19. **filloval()** fills a color oval
20. **fillrect()** fills a color rectangle
21. **fillrect3d()** fills a color 3D rectangle
22. **refresh_vga_manu()** refreshes VGA by paint()
23. **refresh_vga_auto()** refreshes VGA automatically
24. **key_event_ena()** enables keyboard interrupt
25. **key_event_dis()** disables keyboard interrupt

Interrupt Supported in AsmSim

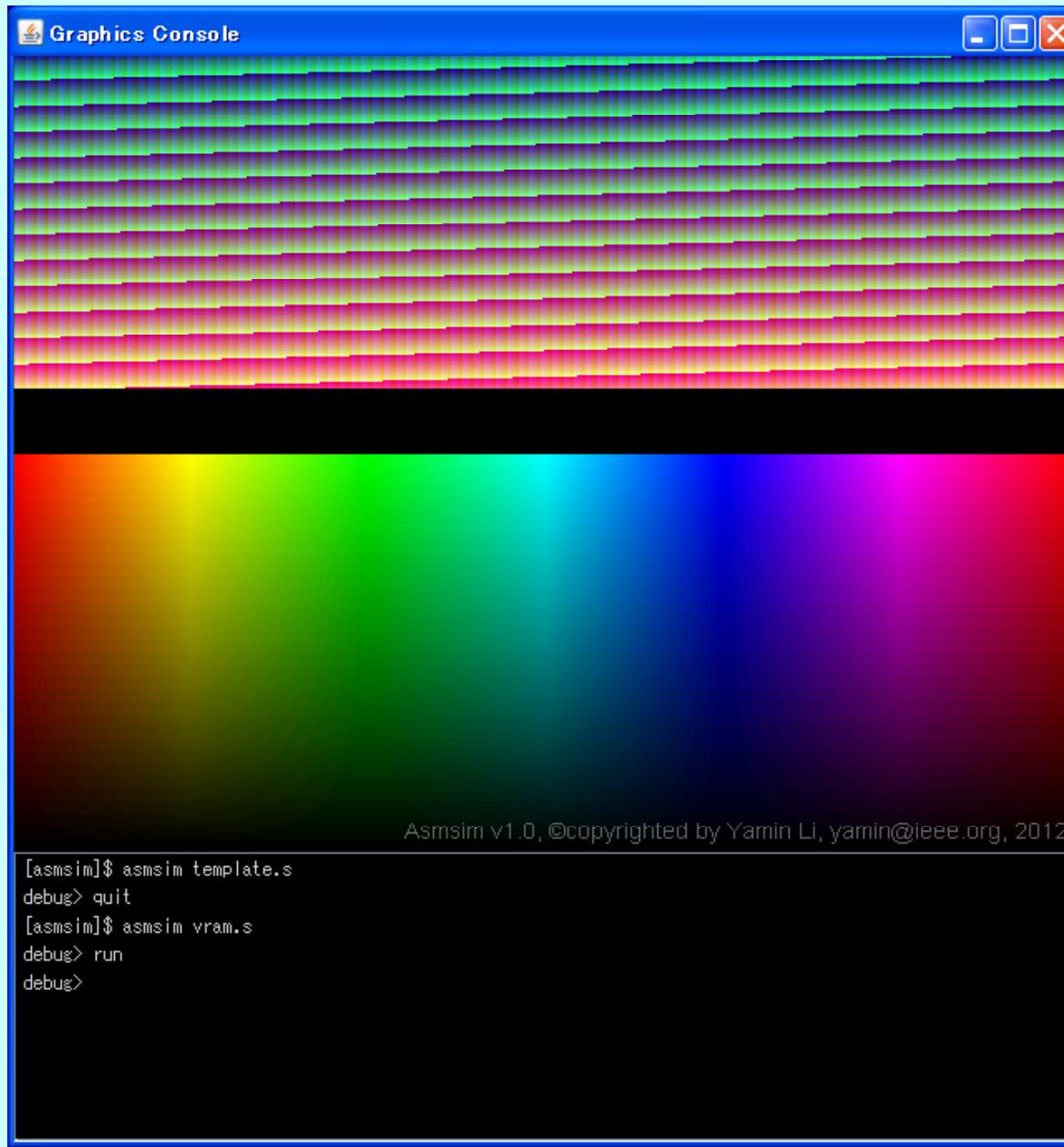


Before transferring to __Key_Event, $EPC \leftarrow$ return address
eret (return from exception): $PC \leftarrow EPC$

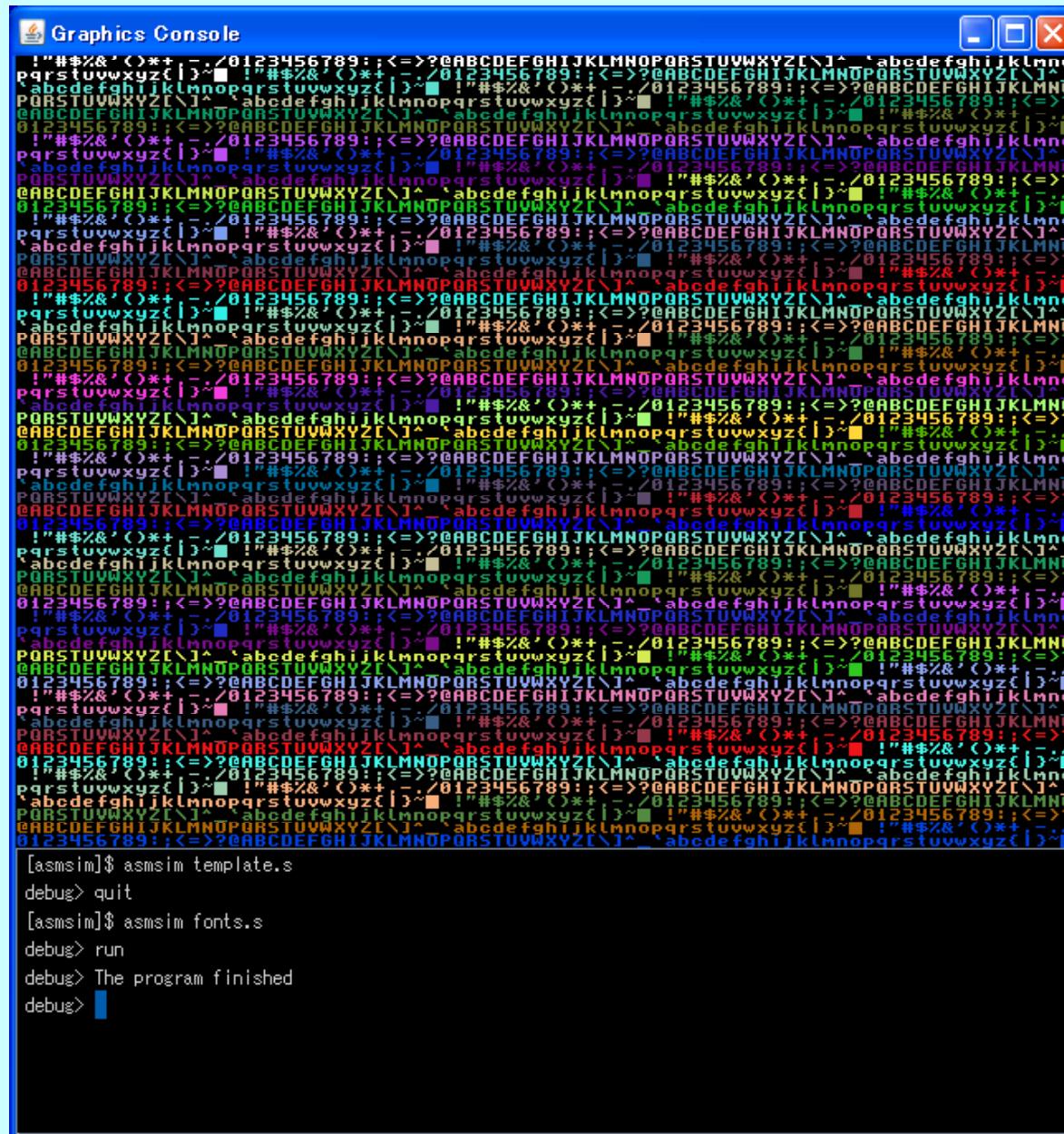
The Graphics Output Example (graphics.s)



The Graphics Output Example (vram.s)



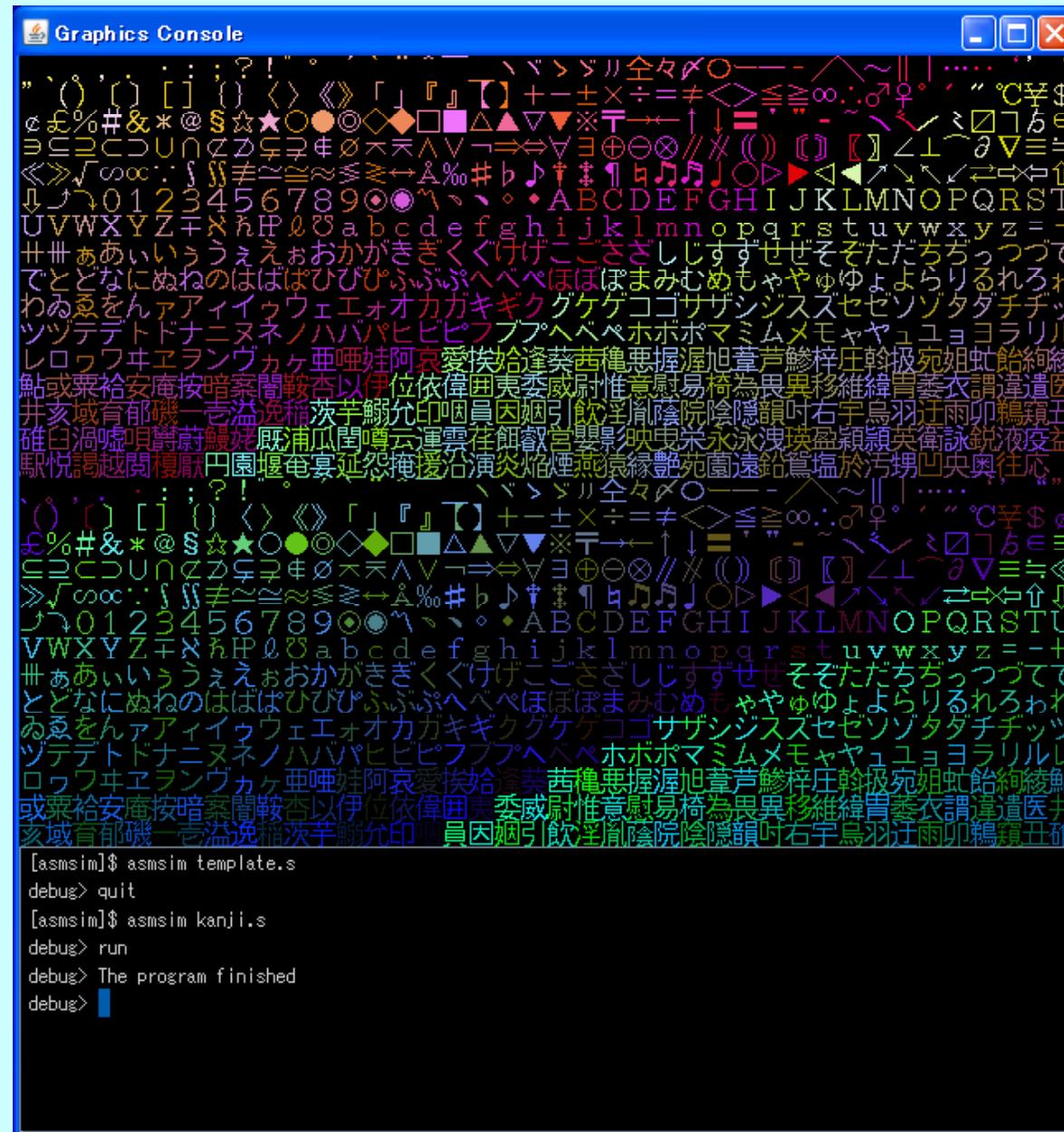
The Graphics Output Example (fonts.s)



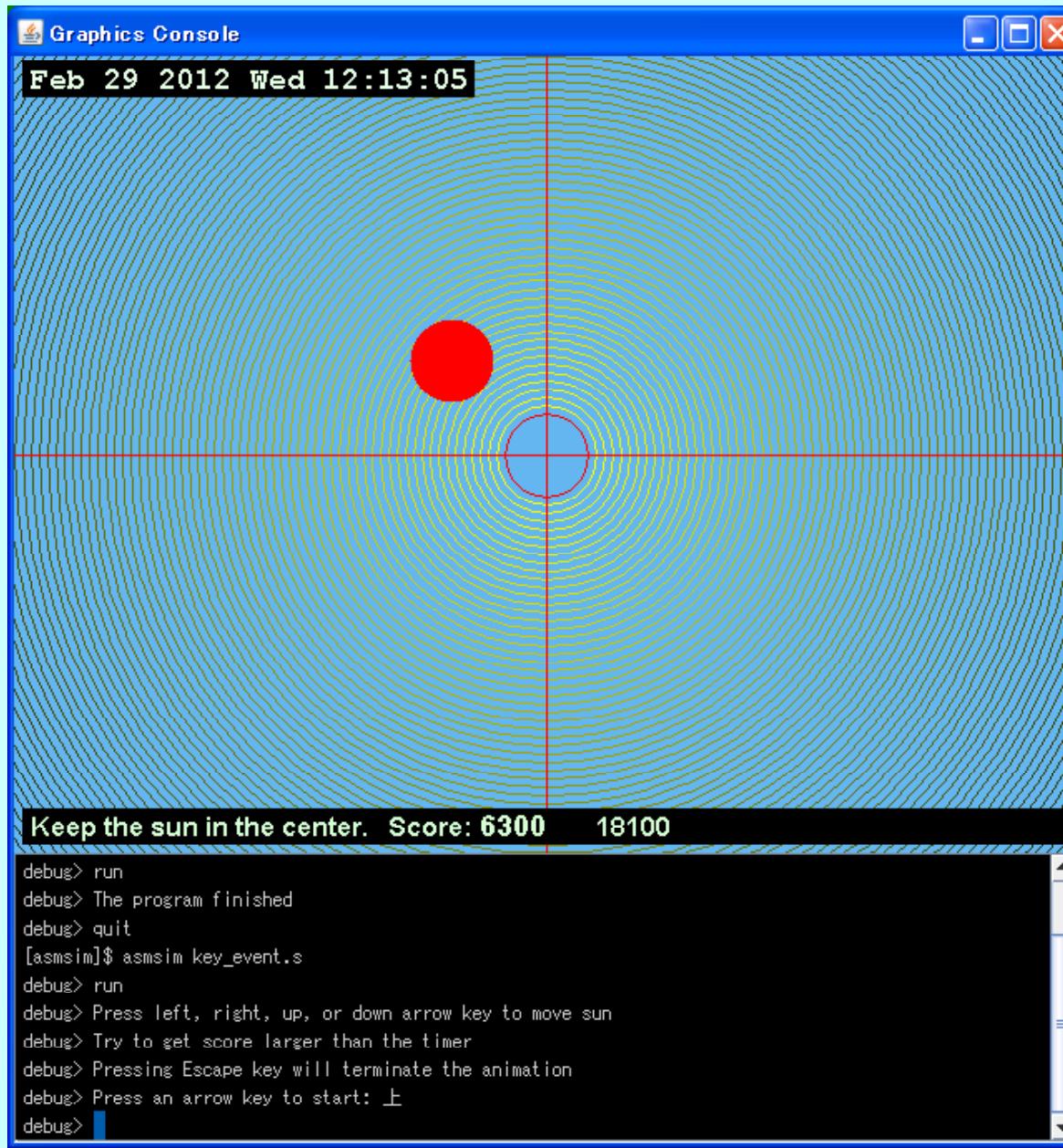
The screenshot shows a terminal window titled "Graphics Console". The window contains a large amount of assembly code, specifically the file "fonts.s", which is used for font rendering. The code is written in a highly compressed and repetitive style, using labels like ".L1", ".L2", etc., and various assembly instructions such as LDI, ST, LD, ADD, SUB, J, and JSR. At the bottom of the terminal window, there is a command-line interface where the user has typed several commands:

```
[asmssim]$ asmsim template.s
debug> quit
[asmssim]$ asmsim fonts.s
debug> run
debug> The program finished
debug>
```

The Graphics Output Example (kanji.s)



The Graphics Output Example (key_event.s)



Xilinx COE Window

Altera MIF Window

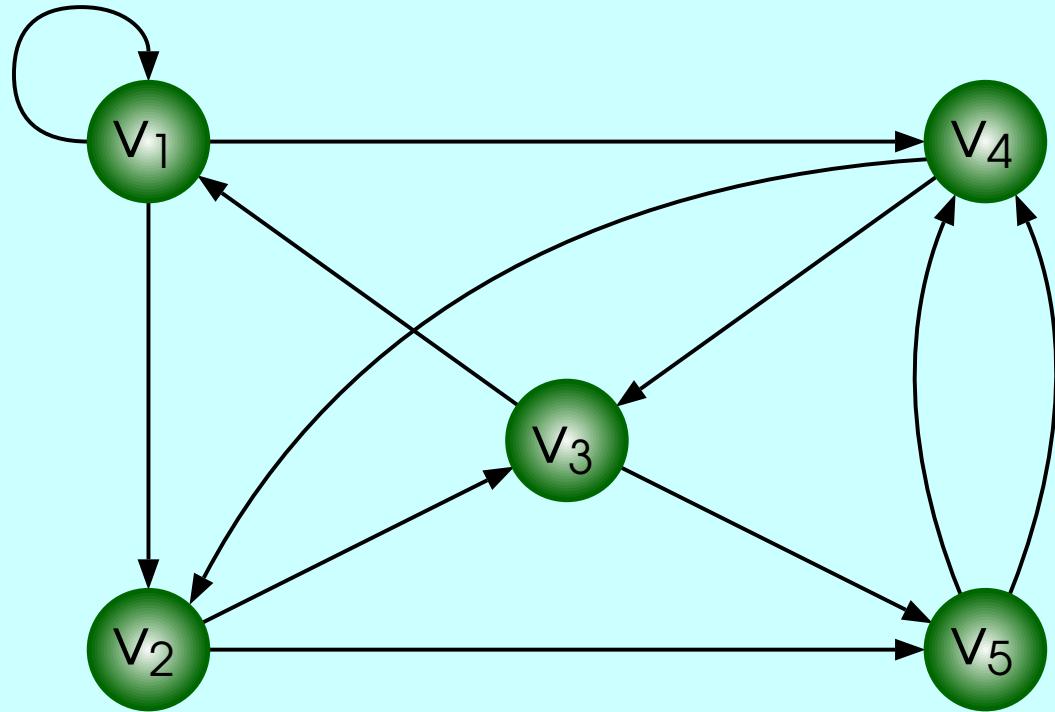
The screenshot shows the Altera MIF window with the following content:

```
% Do not use the syscall functions provided in this simulator.  
% single memory module:  
DEPTH = 32;                      % Memory depth and width are required  
WIDTH = 32;                       % Enter a decimal number  
ADDRESS_RADIX = HEX;               % Address and value radices are optional  
DATA_RADIX = BIN;                 % Enter BIN, DEC, HEX, or OCT; unless  
                                  % otherwise specified, radices = HEX  
CONTENT  
BEGIN  
0000 : 001000111011110111111111000000;  
0001 : 10101111011111000000000011100;  
0002 : 101011110111110000000000111000;  
0003 : 00000000001110111110000000000000;  
0004 : 00000000000000000000000000000000;  
0005 : 00000000001111011101000000000000;  
0006 : 100011110111110000000000111000;  
0007 : 100011110111110000000000111100;  
0008 : 0010001110111101000000001000000;  
0009 : 000000111110000000000000000000001000;  
000a : 00000000000000000000000000000000000000;  
...
```

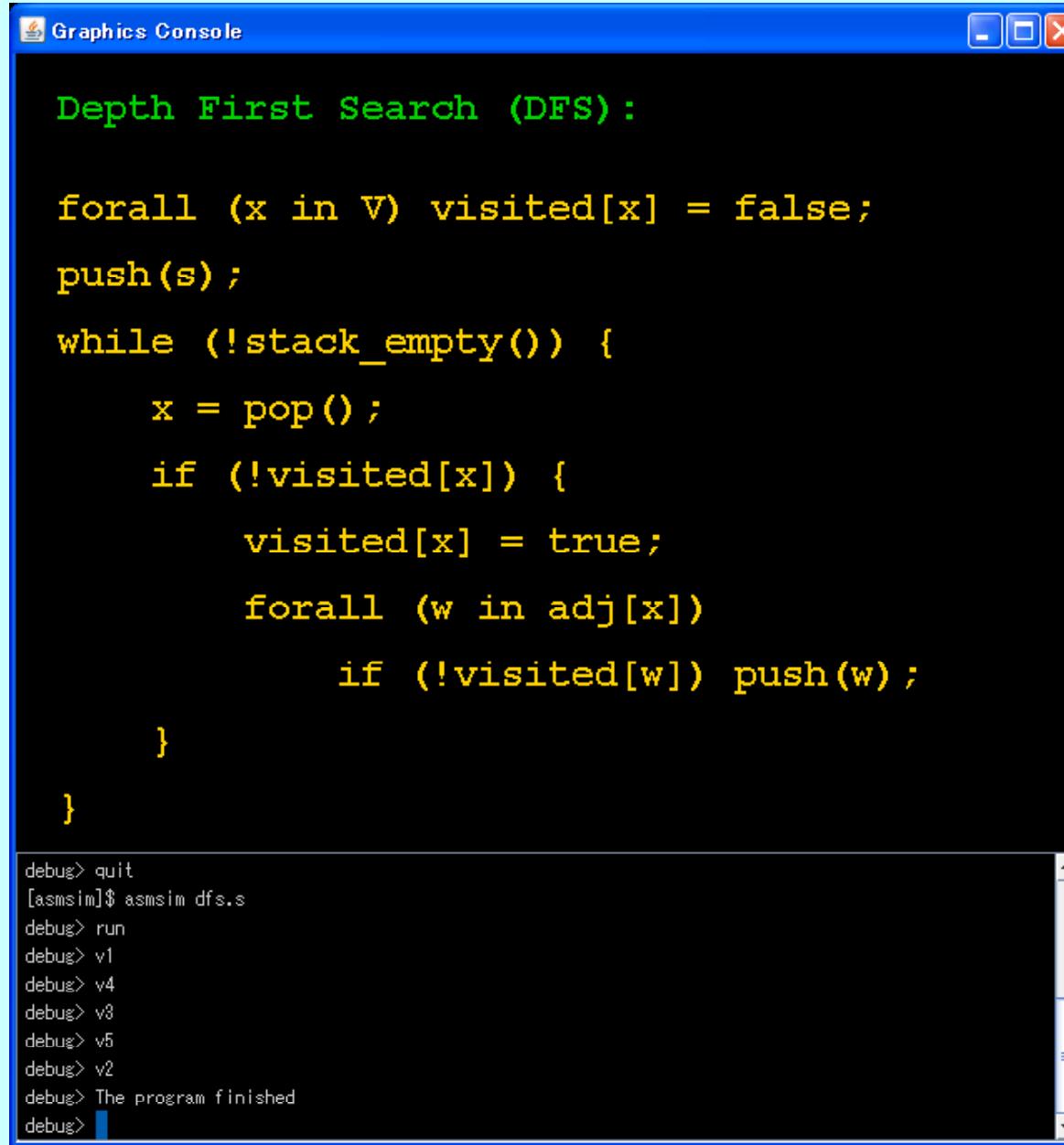
The window has standard OS X-style controls (minimize, maximize, close) and scroll bars on the right side.

Support for Linked-List Data Structure

```
'define NULL 0x0
.data
v1: .word 1, p1
p1: .word v1, p2
p2: .word v2, p3
p3: .word v4, NULL
v2: .word 2, p4
p4: .word v5, p5
p5: .word v3, NULL
v3: .word 3, p6
p6: .word v1, p7
p7: .word v2, p8
p8: .word v5, NULL
v4: .word 4, p9
p9: .word v5, p10
p10: .word v3, NULL
v5: .word 5, p11
p11: .word v4, NULL
```



Depth First Search with Stack (dfs.s)



The image shows a Windows-style "Graphics Console" window titled "Graphics Console". Inside, there is a code editor pane containing C-like pseudocode for Depth First Search (DFS) using a stack. Below it is a terminal-like pane showing the execution of the program.

```
Depth First Search (DFS) :

forall (x in V) visited[x] = false;
push(s);
while (!stack_empty()) {
    x = pop();
    if (!visited[x]) {
        visited[x] = true;
        forall (w in adj[x])
            if (!visited[w]) push(w);
    }
}
```

debug> quit
[asmsim]\$ asmsim dfs.s
debug> run
debug> v1
debug> v4
debug> v3
debug> v5
debug> v2
debug> The program finished
debug>

Breadth First Search with Queue (bfs.s)

The image shows a Windows Graphics Console window titled "Graphics Console". The main area displays the assembly code for Breadth First Search (BFS) using a queue. Below the code, the console shows the command-line interface for running the program.

```
Breadth First Search (BFS) :

forall (x in V) visited[x] = false;
enqueue(s);

while (!queue_empty()) {
    x = dequeue();
    if (!visited[x]) {
        visited[x] = true;
        forall (w in adj[x])
            if (!visited[w]) enqueue(w);
    }
}

debug> quit
[asmsim]$ asmsim bfs.s
debug> run
debug> v1
debug> v4
debug> v2
debug> v3
debug> v5
debug> The program finished
debug>
```

To Let Copy & Paste Work

- Find “`java.policy`” file. It is usually located at
`C:\Program Files\Java\jre6\lib\security\`
- Edit it: Add the following 3 lines in the top of the file:

```
grant codeBase "http://cis.k.hosei.ac.jp/-" {  
    permission java.awt.AWTPermission "accessClipboard";  
};
```

- Save it as “`.java.policy`” at your home directory (fold)
(Note that this filename starts with a dot)
- Close the Web Browser and open it again
- You should have Copy & Paste enabled now

Conclusions

- The assembly language programming is an important way to understanding how the computer works at the machine level
- AsmSim supports
 - System calls like printf() and scanf() in C
 - Graphics draws and fills
 - VRAM and VGA accesses
 - Keyboard interrupt mechanism
 - Linked list data structure
 - Automatic generation of the memory initialization files for Xilinx and Altera FPGAs
- AsmSim can be used by anyone anywhere anytime

