

HUMMUS-N-CHIPS

THE HUMMUS AND CHIPS COLLECTION OF
COMPILER, ASSEMBLER, AND SIMULATOR

HummusPlus Language Specifications

Amlesh Sivanantham

May 10, 2017

HummusPlus Language Specifications

Amlesh Sivanantham

INSTRUCTION	OPCODE	PSUEDO-CODE	DESCRIPTION
HALT	0 0 0 0	<code>exit()</code>	Ends the program.
SHFF	0 0 0 1	<code>PC += unsigned(ARG)</code>	Moves the Program Counter Forward
SHFB	0 0 1 0	<code>PC -= unsigned(ARG)</code>	Moves the Program Counter Backward
BNR	0 0 1 1	<pre> if RS: PC += signed(ARG) </pre>	If the result is not zero, add the signed number to the Program Counter, else do nothing.
INP	0 1 0 0	<pre> if unsigned(ARG): B2 = input() else: B1 = input() </pre>	If the value of ARG is not zero, store the user input into B2, else store it into B1.
STR	0 1 0 1	<pre> if unsigned(ARG): B2 = RS else: B1 = RS </pre>	If the value of ARG is not zero, store the value of RESULT into B2, else store it into B1.
LDB1	0 1 1 0	<code>B1 = Mem[ARG]</code>	Take the byte from location ARG in Data Memory and store it into B1.
LDB2	0 1 1 1	<code>B2 = Mem[ARG]</code>	Take the byte from location ARG in Data memory and store it into B2.
ADDB1	1 0 0 0	<code>RS=B1+unsigned(ARG)</code>	Add the unsigned value of ARG to register B1 and store it into RESULT.
ADDB2	1 0 0 1	<code>RS=B2+unsigned(ARG)</code>	Add the unsigned value of ARG to register B2 and store it into RESULT.
BOOL	1 0 1 0	...	Perform a Boolean operation on B1 and B2 based on ARG and store it into RESULT. More details below.
ADD	1 0 1 1	...	Perform a Addition operation on B1 and B2 based on ARG and store it into RESULT. More details below.
SUBB1	1 1 0 0	<code>RS=B1-unsigned(ARG)</code>	Subtract the unsigned value of ARG from register B1 and store it into RESULT.
SUBB2	1 1 0 1	<code>RS=B2-unsigned(ARG)</code>	Subtract the unsigned value of ARG from register B2 and store it into RESULT.
STM	1 1 1 0	<code>Mem[ARG] = RS</code>	Store the value of B1 or B2 in location RESULT in Data Memory.
MEM	1 1 1 1	...	Handle clearing of Memory or Dynamic storing or reading of Data Memory based on value of ARG.
ADDITION MENT	ARGU-	PSUEDO-CODE	DESCRIPTION
0 0 0 0		<code>RS = B1 + B2</code>	Add B1 and B2.
0 1 0 0		<code>RS = B1 - B2</code>	Subtract B2 from B1.
1 0 0 0		<code>RS = -B1 + B2</code>	Subtract B1 from B2.
1 1 0 0		<code>RS = -B1 - B2</code>	Subtract B2 from Negative B1.

BOOLEAN ARGUMENT	PSUEDO-CODE	DESCRIPTION
0 0 0 0	RS = B1 AND B2	Perform a bitwise AND on B1 and B2 and store it into RESULT.
0 0 0 1	RS = B1 L-AND B2	Perform a logical AND on B1 and B2 and store it into RESULT.
0 0 1 0	RS = B1 OR B2	Perform a bitwise OR on B1 and B2 and store it into RESULT.
0 0 1 1	RS = B1 L-OR B2	Perform a logical OR on B1 and B2 and store it into RESULT.
0 1 0 0	RS = B1 ^ B2	Perform a bitwise XOR on B1 and B2 and store it into RESULT.
0 1 0 1	RS = B1 ^^ B2	Perform a bitwise XNOR on B1 and B2 and store it into RESULT.
0 1 1 0	RS = B1 << 1	Bitshift to the left by 1 on B1 and store it into RESULT.
0 1 1 1	RS = B2 << 1	Bitshift to the left by 1 on B2 and store it into RESULT.
1 0 0 0	RS = ~(B1 AND B2)	Perform a bitwise NAND on B1 and B2 and store it into RESULT.
1 0 0 1	RS = ~(B1 L-AND B2)	Perform a logical NAND on B1 and B2 and store it into RESULT.
1 0 1 0	RS = ~(B1 OR B2)	Perform a bitwise NOR on B1 and B2 and store it into RESULT.
1 0 1 1	RS = ~(B1 L-OR B2)	Perform a logical NOR on B1 and B2 and store it into RESULT.
1 1 0 0	RS = ~B1	Perform a bitwise NOT on B1 and store it into RESULT.
1 1 0 1	RS = ~B2	Perform a bitwise NOT on B2 and store it into RESULT.
1 1 1 0	RS = B1 >> 1	Bitshift to the right by 1 on B1 and store it into RESULT.
1 1 1 1	RS = B2 >> 1	Bitshift to the right by 1 on B2 and store it into RESULT.
MEMORY ARGUMENT	PSUEDO-CODE	DESCRIPTION
0 0 0 0	For all Mem, Mem = 0	Clear the data memory.
0 0 0 1	B1 = Mem[B1]	Take byte from location B1 and store it into B1.
0 0 1 0	B1 = Mem[B2]	Take byte from location B2 and store it into B1.
0 0 1 1	B1 = Mem[RS]	Take byte from location RESULT and store it into B1.
0 1 0 0	Mem[B1] = B1	Store value of B1 in address B1 in Data Memory.
0 1 0 1	B2 = Mem[B1]	Take byte from location B1 and store it into B2.
0 1 1 0	B2 = Mem[B2]	Take byte from location B2 and store it into B2.
0 1 1 1	B2 = Mem[RS]	Take byte from location RESULT and store it into B2.
1 0 0 0	Mem[B1] = B2	Store value of B2 in address B1 in Data Memory.
1 0 0 1	Mem[B2] = B1	Store value of B1 in address B2 in Data Memory.
1 0 1 0	Mem[B2] = B2	Store value of B2 in address B2 in Data Memory.
1 0 1 1	Mem[B2] = RS	Store value of RESULT in address B2 in Data Memory.
1 1 0 0	Mem[B1] = RS	Store value of RESULT in address B1 in Data Memory.
1 1 0 1	Mem[RS] = B1	Store value of B1 in address RESULT in Data Memory.
1 1 1 0	Mem[RS] = B2	Store value of B2 in address RESULT in Data Memory.
1 1 1 1	Mem[RS] = RS	Store value of RESULT in address RESULT in Data Memory.

Example of the language can be seen below. This is the implementation of a Turing Machine.

```
# INITIALIZE THE UNIVERSAL TURING MACHINE
# Setup the vairables.
# Assume all reserved memory and
# registers are zero.
```

```
#####
```

```
# Specify the starting address
# of the tape.
ADDB1    8      {INITIALIZE_MACHINE}
STR      B1
BOOL     B1<<1
STM      TAPE_ADDR
BOOL     B1<<1
STR      B1
BOOL     B1<<1
STR      B1
BOOL     B1<<1
STR      B1
# This is the max tape address and the
# value of the initial state
STM      TAPE_ADDR_END
STM      NEXT_STATE
```

```
# Make sure we read the first value of
# the tape instead of skipping it.
ADDB1    2
MEM      B2<-Mem[RS]
LDB1     TAPE_ADDR
ADD      B1-B2
STM      TAPE_ADDR
```

```
# Move to the next block
SHFF     {UPDATE_THE_STATE}
```

```
#####

# UPDATE THE STATE VARIABLES

# Update the current state
LDB1    NEXT_STATE {UPDATE_THE_STATE}

# Update the value of the tape
# we are searching for
ADDB1   0
MEM     B2<-Mem[RS]
ADDB2   0
STM     SEARCH_FOR

# Update the value to replace with
ADDB1   1
MEM     B2<-Mem[RS]
ADDB2   0
STM     REPLACE_WITH

# Update the tape traversal direction
ADDB1   2
MEM     B2<-Mem[RS]
ADDB2   0
STM     TAPE_DIR

# Update the next state
ADDB1   3
MEM     B2<-Mem[RS]
ADDB2   0
STM     NEXT_STATE

SHFF    {START_TRAVERSING_TAPE}

#####

# TRAVERSE THE TAPE
```

```

# Increment the address
# Load the current address and
# the increment value
LDB1    TAPE_ADDR    {START_TRAVERSING_TAPE}
LDB2    TAPE_DIR

# Verify that the machine has not halted.
# i.e. increment value is zero
ADDB2    0
BNR      {UPDATE_ADDRESS}
HALT

# Actually increment the address here.
ADD      B1+B2        {UPDATE_ADDRESS}
STM      TAPE_ADDR

# load the memory of the tape
# and what state is looking for
MEM      B1<-Mem[RS]
LDB2     SEARCH_FOR

# evaluate if it is what we are
# looking for and branch accordingly
ADD      B1-B2
BNR      {FOUND_ON_TAPE}
SHFB     {START_TRAVERSING_TAPE}

#####

# IF FOUND THE ITEM ON TAPE,
# UPDATE THE CURRENT CELL

LDB1     TAPE_ADDR    {FOUND_ON_TAPE}
LDB2     REPLACE_WITH
MEM      Mem[B1]<-B2
SHFB     {UPDATE_THE_STATE}

```