HRAM0 Model: Specification

Michael Clear

Document version: 0.1
May 31, 2022

1 Overview

This document formally specifies a Heap Random Access Machine (HRAM) model called HRAM0. A HRAM is a computational model which is a simple RAM model with additional instructions to facilitate allocation and deallocation of memory blocks. The memory contains an unbounded number of storage locations of unbounded size. HRAM0 is one such model whose operation and semantics we elucidate in this document.

1.1 Notation

For some set S, we use the notation $\mathbf{s} := [s_1, \ldots, s_k]$ for a finite sequence $\mathbf{s} \in S^*$ of length k where $s_1, \ldots, s_k \in S$. Sequence variables are written in boldface. The length of a sequence $\mathbf{s}$ is denoted by $|\mathbf{s}|$. The subsequence of some sequence $\mathbf{s}$ consisting of those contiguous elements from index i (inclusive) to index $j > i$ (exclusive) is denoted by $\mathbf{s}[i;j]$; that is, we have $\mathbf{s}[i;j] = [s_i, s_{i+1}, \ldots, s_{j-1}]$. When we wish to refer to the subsequence of $\mathbf{s}$ starting at index i and continuing to the end of the sequence, we use the notation $\mathbf{s}[i;\ldots]$, which is shorthand for $\mathbf{s}[i;|\mathbf{s}|] = [s_i, s_{i+1}, \ldots, s_{|\mathbf{s}|-1}]$.

2 Architecture and Instruction Set

The memory contains an unbounded number of storage locations of unbounded size. It comprises two separate *segments*: a code segment and a data segment, both of which are addressed starting with zero. The code segment is read-only. The machine supports a finite number of "registers" of unbounded size. There are ρ "data registers", where ρ is a machine parameter, which are written r0, r1, r2 etc. Also there are two special registers: (1) pc which contains the program counter (address of the next instruction) and which is assigned the number -2 and (2). n which is the input length and which is assigned the number -1. For the remainder of this document, for ease of presentation, we assign the pc register the number ρ and the n register the number $\rho + 1$. By abuse of terminology, we use the term "word“ to refer to the contents of a memory location (i.e. at a particular address) or the contents of a register despite such storage locations being unbounded with the understanding that any concrete instantiation of the machine would impose some size w (in bits) and therefore serve as a *word* in the traditional sense.

An instruction with m operands is encoded in $m + 1$ successive words in the code segment, the first word is the opcode and each subsequent word is a distinct operand. The instruction set is as follows:

Operation	Mnemonic	Opcode	Description
Halt	HLT	0	Stop
Put	PUT $\langle\mathsf{constant}\rangle, \langle\mathsf{reg}\rangle$	1	Put constant value in a register
Add	ADD $\langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle$	2	Add first two registers and store the result in the third register
Subtract	SUB $\langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle$	3	Subtract first register from second register and store the result in the third register
Load	LOD $\langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle$	4	Load word from memory at address given by first register into second register
Store	STO $\langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle$	5	Store word from first register to memory address given by second register
Branch if negative	BRN $\langle\mathsf{reg}\rangle, \langle\mathsf{constant}\rangle$	6	If contents of first register is negative, jump to non-negative constant address given as the second operand; ELSE next instruction
Call subroutine	CAL $\langle\mathsf{constant}\rangle$	7	Jump to non-negative constant address and record address of next instruction which can be returned to with RET (see next)
Return from subroutine	RET	8	Return to the instruction following most recent CAL. If no such CAL, then halt.
Allocate (malloc)	MAL $\langle\mathsf{reg}\rangle, \langle\mathsf{reg}\rangle$	9	Allocate block of memory of size (in words) given by first register and store the address in the second register
Deallocate (free)	FRE $\langle\mathsf{reg}\rangle$	10	Free block of memory that starts at the address given by the first register

A program is loaded into the code segment at address zero and the input is loaded into the data segment at address zero. If there is statically-allocated data, the input is loaded after that in the data segment. Here we assume the static data is empty. All remaining memory locations are initialized to zero on machine start-up as are all registers. Suppose the program occupies c words of the code segment and the static data occupies d words of the data segment. Then the region where blocks of memory are allocated, known as the *heap*, begins at address d (data segment). When the machine starts, control is passed to the instruction whose encoding begins at address zero in the code segment. Our choice of opcode 0 for the halt instruction is natural given that all words in the code segment from address c onwards are initialized to zero. No provision is given for a stack, it must be setup and maintained by the programmer by suitably allocating a sufficiently large block of memory.

Programs use *labels* that correspond to specific addresses in the code segment and whose absolute value can be readily computed from the base address of the program and its offset determined by the number of words occupied by the preceding instructions. Typically the presentation of programs do not specify a base address and therefore the labels correspond to relative offsets until the program is loaded into memory where they are appropriately translated with respect to some base address. An exception to this is the first program we present which is located at address zero to which control is passed by the machine on startup. We employ the label *start* to correspond to address zero. This particular program initializes the *global* environment that is used by our later programs. For example, it reserves register r2 for the constant -1.

Next we provide a more substantive example, namely a subroutine that (naively) multiplies two integers residing in r0 (multiplicand) and r1 (multiplier) and places the result in r0. We employ a very basic calling convention. A more powerful realization of subroutines can be achieved by allocating a region of memory to serve as a stack and reserving a register for the stack pointer; then arguments can be pushed to the stack and registers can be saved and later restored to avoid unintended side-effects. On the other hand, the basic approach we take here affords illustrative simplicity and comparative brevity wherein the arguments are placed in registers r0, r1 etc... and r0 is written with the return value (which suffices for example for integer-valued functions) but some registers are “dirtied“, but nevertheless, this basic approach suffices here for our purpose of illustration. Figure 1 also declares a “global variable” named status which is a single integer that is initialized to 0. Therefore the static data $\mathbf{d}$ for this program is $\mathbf{d} := [0]$.

```
Data @ base 0:
    1 status := 0 // 1 integer element, initialized to 0

Code @ base 0:
    start:
       PUT −1, r2
       BRN r2, main // jump to main
```

Figure 1. A program P_{init} that runs on startup and initializes register r2 for convenient usage by other code.

```
Subroutine mult_naive:
    mult_naive:
       PUT 0, r3
       ADD r0, r3, r3 // multiplicand to r3
       PUT 0, r0 // r0 is now for result
       BRN r1, negloop
    posloop:
       ADD r2, r1, r1 // r1 is multiplier
       BRN r1, multend
       ADD r3, r0, r0
       BRN r2, posloop
       BRN r2, multend
    negloop:
       SUB r3, r0, r0
       SUB r2, r1, r1
       BRN r1, negloop
multend:
       RET // jump to return address
```

Figure 2. A subroutine to naively multiply two integers (the multiplicand in register r0 and the multiplier in r1) and store the result in r0.

Macros A useful tool we now describe is the notion of a macro. A macro associates with a name and list of arguments a sequence of target instructions containing placeholders that each reference a specific argument. Each occurrence of the macro name (and list of arguments) in an assembly specification is “expanded“ to (i.e. replaced with) the sequence of target instructions wherein each placeholder is substituted for the argument it references. Hence, macros are somewhat akin to inline function calls in C. Therefore, repetitive instruction patterns can be captured by a macro resulting in an abridged expression in assembly language. Below is an example of a macro; it effectively builds a "branch if zero" "instruction". It assumes that the register r2 contains the constant -1.

- **BRZ**(r, target):
  - ∗ BRN r, notzero
    ∗ ADD r2, r, r // r -= 1
    ∗ BRN r, iszero
    ∗ SUB r2, r, r // restore r to original
    ∗ BRN r2, notzero
    iszero:
  - ∗ PUT 0, r // restore r to original (i.e. 0)
    ∗ BRN r2, target
  - notzero: // to next instruction

Input/Output Input and output are handled by reading and writing respectively to the data segment. The program's input is located directly after the static data. The register n contains the length of the input. We continue our example program from above by specifying the program body (main) below. Recall that our static data consists of a single integer (the variable *status*) which will be located at address 0. Hence, the program input begins at address 1. The program below consists of reading input, namely loading multiplicand from address 1 and multiplier from address 2, invoking subroutine mult_naive and writing the returned result to address 1. We first check however that the input length $n \geq 2$. If this is not the case, we set the status variable to -1 by storing -1 (i.e. the contents of r2) to address 0. By default, the value of status at address 0 is 0, Therefore, a status of 0 indicates a successful outcome (i.e. sufficient input is provided and a result is computed) whereas -1 indicates a "failure" outcome (i.e. insufficient input has been provided).

Halting States We distinguish two distinct terminating states: HALT and ERROR:

- HALT: This is the state of termination triggered by the HLT instruction and signifies a "graceful" termination. When this state is reached, it can be concluded that no unsafe memory access has occurred.
- ERROR: This is the state of termination triggered by the detection of an unsafe memory access such as a load or store to an address a such that $a \geq d$ and a is not within the bounds of a block of memory allocated by MAL or such a block if it exists has been marked deallocated by FRE.

Program Body (main):

```
main:
  PUT 1, r3
  SUB n, r3, r0
  BRN r0, multiply
insufficient_input:
  PUT 0, r4
  STO r2, r4 // write status of -1 to address 0
  BRN r2, end
multiply:
  LOD r3, r0 // load multiplicand (recall r3=1)
  PUT 2, r4
  LOD r4, r1 // read multiplier
  CAL mult_naive // call mult_naive
  STO r0, r3 // write result to address 1
end:
```

Figure 3. Read input, call subroutine mult_naive and output result.

3 Formal Model

We begin by describing the program state ps which is contained within the larger machine state ms as a substructure. The program state ps is a tuple ps $:= (\mathbf{r}, M, B, e, \mathbf{s})$ where $\mathbf{r} \in \mathbb{Z}^{\rho+2}$ is a $\rho + 2$-dimensional integer vector, and $M : \mathbb{Z} \to \mathbb{Z}$ and $B : \mathbb{Z} \to \mathbb{Z}$ are partial functions, each represented as a relation in the form of a set of ordered pairs, $e \in \mathbb{Z}$ is an integer, and $\mathbf{s}$ is a sequence. In relation to the vector $\mathbf{r}$, component r_0 represents register r0, component r_1 represents register r1 etc... from 0 to $\rho - 1$ (inclusive) and component r_ρ represents the special register pc (program counter) and component $r_{\rho+1}$ represents register n (input length). For an address a, if $M(a)$ is defined, the partial function M maps address a to the value $M(a)$ stored at that address in memory (data segment). A pair $(a, b) \in B$ represents an allocated block with starting address a containing the range of addresses between a and b (exclusive); in such a case, $B(a)$ is defined and maps to the end address b. The integer e determines the starting address of the next block to be allocated. Finally, $\mathbf{s}$ is a sequence that is used as a stack; that is, items are popped from the back and pushed to the back. It is used to record the return address of each CAL instruction and therefore allows arbitrary nesting of subroutine calls. The set of all program states is denoted by $\mathcal{PS}$.

Let $\Sigma = \{$START, LOAD, RUN, WAIT, HALT, ERROR,
LOADC, LOADD, LOADI, FETCH, DECODE,

$\mathsf{EXEC\_HLT}, \mathsf{EXEC\_PUT}, \mathsf{EXEC\_ADD}, \mathsf{EXEC\_SUB},$
$\mathsf{EXEC\_LOD}, \mathsf{EXEC\_STO}, \mathsf{EXEC\_BRN}, \mathsf{EXEC\_CAL}, \mathsf{EXEC\_RET}, \mathsf{EXEC\_MAL}, \mathsf{EXEC\_FRE}\}$
be a set of "action" states indicating the current stage/phase of execution of the machine. The machine state ms is defined as $\mathsf{ms} := (\sigma, \mathbf{c}, \mathbf{d}, \mathbf{t}, \mathsf{ps})$ where $\sigma \in \Sigma$ is the "action" *state*, $\mathbf{c} \in \mathbb{Z}^*$ is the contents of the code segment, $\mathbf{d} \in \mathbb{Z}^*$ is the *initial* contents of the data segment, $\mathbf{t} \in \mathbb{Z}^3$ is a vector of 3 integers whose purpose will become clear later, and $\mathsf{ps} \in \mathcal{PS}$ is the program state.

In addition to the memory (code and data segments) and the registers, there is also an input *tape* of unbounded length which is used by the meta-execution subsystem for specifying the programs and data to be *loaded* into memory in addition to specifying the program input. Note that the program input is strictly non-interactive; it is pre-loaded into memory prior to the execution of the program, which accesses the program input from memory. The program can obtain the length of the program input (in "words") from the register n, designated register number $\rho+1$. Therefore, the input length is embedded in the program state as $r_{\rho+1}$ (i.e. component number $\rho+1$ of the vector $\mathbf{r}$, indexed from 0).

The aforementioned tape can be moved left, moved right or remain unchanged at each execution step of the machine. Our description in this section considers an abstraction of the *actual* tape. We abstract away the fact that only a finite set of symbols can be written to the tape ($\{0, 1, \perp\}$) and a finite state sub-machine decodes an arbitrary-length string of bits terminated with $\perp$ into the register x. This concrete representation and associated decoding process is deferred to a later version, presenting a higher-level view here that consequently allows us to develop a simplified formalism. In a nutshell, in our abstracted view, the register $x \in \mathbb{Z} \cup \{\perp\}$ is either an integer value or the symbol $\perp$ which means end of tape.

3.1 Formal Semantics

The formal semantics of the instruction set and precise operation of the machine are given by a state transition function $\delta : \mathcal{MS} \times (\mathbb{Z} \cup \{\perp\}) \to \mathcal{MS} \times \{\leftarrow, -, \rightarrow\}$ that maps a machine state and tape element pair to a new machine state and direction symbol. The direction symbol is one of the following: move left ($\leftarrow$), remain in place ($-$) or move right ($\rightarrow$).

State σ	Input x	Action **in:** $(\mathsf{ms} := (\sigma, \mathbf{c}, \mathbf{d}, \mathbf{t}, \mathsf{ps} := (\mathbf{r}, M, B.e, \mathbf{s})), x)$
EXEC_HLT	*	**out:** $(\mathsf{ms}' := (\mathsf{HALT}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}), -)$
EXEC_PUT	*	$r'_{t_1} \leftarrow t_0$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq t_1$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}'.M, B.e, \mathbf{s})), -)$
EXEC_ADD	*	$r'_{t_2} \leftarrow r_{t_0} + r_{t_1}$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq t_2$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}'.M, B.e, \mathbf{s})), -)$
EXEC_SUB	*	$r'_{t_2} \leftarrow r_{t_1} - r_{t_0}$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq t_2$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}'.M, B.e, \mathbf{s})), -)$
EXEC_LOD	*	If $M(r_{t_0})$: $\quad r'_{t_1} \leftarrow M(r_{t_0})$ $\quad \sigma' \leftarrow \mathsf{FETCH}$ Else: $\quad r'_{t_1} \leftarrow r_{t_1}$ $\quad \sigma' \leftarrow \mathsf{ERROR}$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq t_1$ **out:** $(\mathsf{ms}' := (\sigma', \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}', M, B.e, \mathbf{s})), -)$
EXEC_STO	*	If $M(r_{t_1})$: $\quad M' \leftarrow (M \setminus \{(r_{t_1}, M(r_{t_1}))\}) \cup \{(r_{t_1}, r_{t_0})\}$ $\quad \sigma' \leftarrow \mathsf{FETCH}$ Else: $\quad M' \leftarrow M$ $\quad \sigma' \leftarrow \mathsf{ERROR}$ **out:** $(\mathsf{ms}' := (\sigma', \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}, M', B.e, \mathbf{s})), -)$
EXEC_BRN	*	If $r_{t_0} < 0$: $\quad r'_\rho \leftarrow t_1$ Else: $\quad r'_\rho \leftarrow r_\rho$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq \rho$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}', M, B.e, \mathbf{s})), -)$
EXEC_CAL	*	$\mathbf{s}' \leftarrow \mathbf{s} \parallel r_\rho$ $r'_\rho \leftarrow t_0$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq \rho$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}', M, B.e, \mathbf{s}')), -)$
EXEC_RET	*	If $\mathbf{s} \neq \varepsilon$: $\quad \mathbf{s}' \parallel a \leftarrow \mathbf{s}$ such that $a \in \mathbb{Z}$ $\quad r'_\rho \leftarrow a$ $\quad \sigma' \leftarrow \mathsf{FETCH}$ Else: $\quad \mathbf{s}' \leftarrow \mathbf{s}$ $\quad r'_\rho \leftarrow r_\rho$ $\quad \sigma' \leftarrow \mathsf{HALT}$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq \rho$ **out:** $(\mathsf{ms}' := (\sigma', \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}', M, B.e, \mathbf{s}')), -)$
EXEC_MAL	*	If $r_{t_0} > 0$: $\quad e' \leftarrow e + r_{t_0} + \zeta$ $\quad B' \leftarrow B \cup \{(e, e')\}$ $\quad M' \leftarrow M \cup \{(i, 0) : e \leq i < e'\}$ $\quad r'_{t_1} \leftarrow e$ Else: $\quad e' \leftarrow e$ $\quad B' \leftarrow B$ $\quad M' \leftarrow M$ $\quad r'_{t_1} \leftarrow r_{t_1}$ $r'_i \leftarrow r_i$ for $0 \leq i \leq \rho + 1, i \neq t_1$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}', M', B'.e', \mathbf{s})), -)$

State σ	Input x	Action **in:** $(\mathsf{ms} := (\sigma, \mathbf{c}, \mathbf{d}, \mathbf{t}, \mathsf{ps} := (\mathbf{r}, M, B.e, \mathbf{s})), x)$
EXEC_FRE	*	If $B(r_{t_0})$: $B' \leftarrow B \setminus \{(r_{t_0}, B(r_{t_0}))\}$ $M' \leftarrow M \setminus \{(i, M(i)) : r_{t_0} \leq i < B(r_{t_0})\}$ Else: $B' \leftarrow B$ $M' \leftarrow M$ **out:** $(\mathsf{ms}' := (\mathsf{FETCH}, \mathbf{c}, \mathbf{d}, \mathbf{0}, \mathsf{ps}' := (\mathbf{r}, M', B'.e, \mathbf{s})), -)$

4 Parameters

An instance of HRAM0 is parameterized by two positive integers:

- ρ - **number of data registers**
- ζ - **inter-block gap size**: This parameter specifies the size of the region between allocated blocks that serves to detect overflows by triggering the error state when a memory location in one of these regions is accessed.

4.1 Standard Reference Model

The standard reference model is a specific parameterization, referred to by the designation HRAM0s, consisting of the parameter selection $\rho = 14$ and $\zeta = 10$. This choice was informed by convenience in some standard programming tasks along with accommodation of many typical overflow scenarios. Recommendation of a particular parameter selection affords ease of comparison across different uses.