Simulation Modelling of Historical Computers

Roland Ibbett & David Dolman

In the early days of computing, when all computers were physically large devices, students taking computing degree courses could be shown which part of the computer was which and could see flashing display lights showing what was going on inside the machine. With the advent of the microprocessor, this form of human-computer interaction became largely a thing of the past. Around the same time, however, bit-mapped graphical display devices appeared and these could be used to demonstrate the inner workings of computers via animation of on-screen models. HASE, a Hierarchical computer Architecture design and Simulation Environment, was developed at the University of Edinburgh with this purpose in mind.

HASE simulation models of a variety of computer architectures and architectural components have been created, some for research investigations of large-scale computing systems and others for use as teaching and learning resources in lectures, for student self-learning or for virtual laboratory experiments. The HASE website (*www.icsa.inf.ed.ac.uk/research/groups/hase/*) provides access to the models and to the Java code for HASE itself. Each model has its own supporting webpages describing the system being modelled, as well as the model itself, and from which the source files for each model can be downloaded. These files can be used as inputs to HASE, which has options to load a project, to compile a simulation executable and to run the simulation. Running a simulation produces a trace file which can then be used to animate the on-screen display of the model to show data movements, parameter value updates, state changes, etc.

Models of five historically significant computers have been created by one of the authors (RI): Atlas, MU5, the CDC 6600, the Cray-1 and, most recently, the Manchester 'Baby' computer. Space precludes describing all of them here, so the interested reader is referred to the HASE website for details of the Cray-1 model. The models are intended to demonstrate the principles of operation of the computers they represent, rather than to be able to execute real programs, so they do not attempt to reproduce all of the more esoteric details of the real machines. They do however try to emulate, as far as possible, the organisation and design features of the corresponding hardware. This has led to further development and enhancement (by DD) of HASE itself.

The HASE Simulation Environment

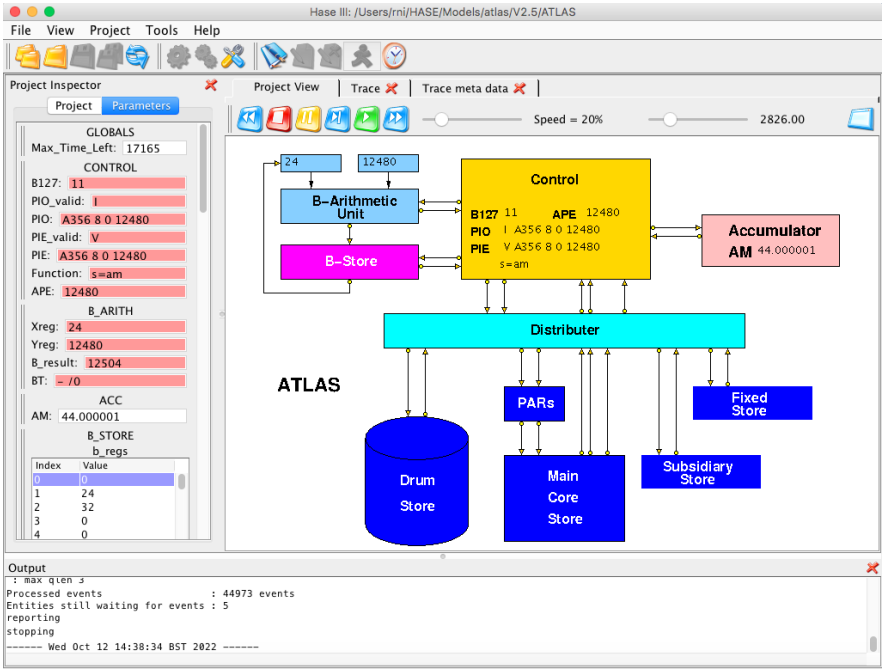


Figure 1 : The HASE GUI showing the Atlas Model

Figure 1 shows a screen image of the HASE Graphical User Interface. The icons in the top row allow the user to load a model, compile it, run the simulation code thus created and to load the trace file produced by running a simulation back into the model for animation. The main Project View pane shows the screen image of the model itself, in this case the Atlas model. The Project Inspector pane displays the values of the parameters defined for the model, while the Output pane displays information being reported back to the user during the various phases of activity involved in a simulation experiment.

Each of the blocks in the image represents an entity within the model: HASE uses an Entity Description Language file (`project.edl`) and an Entity Layout File (`project.edf`) to specify an architecture. A `project.edl` file contains five sections: **`PREAMBLE`**, **`PARAMLIB`**, **`GLOBALS`**, **`ENTITYLIB`** and **`STRUCTURE`**. The **`PREAMBLE`** section contains the name of the project, the name of the author(s), the version number and a description. The **`PARAMLIB`** (Parameter Library) is used to create the type definitions of parameters used by the project. HASE provides a set of built-in types: bool, char, integer, unsigned integer, long, floating-point, string and range, from which more complex types can be created. These include **`ARRAY`** (array parameter type), **`ENUM`** (enumerated type parameter) and **`STRUCT`** (structure parameter), all of which are similar to those found in C and C++. In addition,

HASE provides an **ARRAYI** type, which allows individual instructions within an array of instructions to be labelled, a **LINK** type, used when creating instances of entity ports and links between them, and an **INSTR** type, which can be used to define the instruction set of a processor. The **INSTR** type allows instructions to be grouped together, a useful feature for decoding purposes.

The **ENTITYLIB** section contains the definitions of the entities. Each entity definition includes a type name, a textual description, a list of states, a list of parameters and a list of ports. The **STRUCTURE** section defines the architecture by (a) creating instances of the entities and assigning instance names to each of them, (b) creating links between entity ports. The **GLOBALS** section contains parameters that are accessible by all entities in the model.

Using the data in the **project.edl** file, HASE creates a library of parameter types from which parameter instances (variables) can be created and a library of components from which an architecture can be constructed by linking the defined components together in the desired manner. The screen image of the project is defined by the **project.edf** file which contains screen position information about the displayed entities, their ports and parameters, and the name of the appropriate bitmap icon for each of an entity's possible states. The icons are contained in a separate bitmaps subdirectory.

The behaviour of the entities is coded in Hase++, a discrete event simulation engine with a programming interface similar to that of Sim++, but implemented using C++ and threads. It includes a set of library routines to provide for process oriented discrete event simulation and a run-time system for multi-threading many objects in parallel and keeping track of simulation time. For each entity there is a separate file (**entity.hase**) containing its behavioural simulation code. There is also an option to include a global fns file of user defined functions available to all entities.

The HASE GUI and the HASE model compiler are written in Java. The model compiler generates C++ code which is then compiled using the C++ compiler native to the system on which HASE is being run. This is linked against the HASE Runtime (written in C++) to produce the simulation executable. HASE is supported on OSX, Windows 7/8/10 and Linux.

Modelling Constraints

In any simulation modelling system there is a trade-off between accuracy and performance. HASE was designed primarily as a high-level visualisation tool for computer architecture students and therefore simulates systems at register/word level rather than bit level. This inevitably imposes some limitations on the way models can be constructed, e.g. registers are modelled using typed variables and stores are modelled as arrays of typed variables. This means, for example, that fixed-point (integer) variables, floating-point (real) variables and instructions cannot be held in shared arrays. Furthermore, most early computers used what

now seem like rather idiosyncratic number formats, so achieving a high degree of verisimilitude in the simulation of floating-point operations would add a significant level of complication. In the MU5 and CDC 6600 models, this problem is avoided by simply representing all numbers as integers. In the Atlas model, advantage has been taken of the way the paging mechanism works to separate, unrealistically of course, fixed and floating-point numbers into separate pages; floating-point arithmetic is implemented using the floating-point operations provided by the underlying hardware of the computer on which the simulation is running, accessed via standard C++ operations.

There are also constraints on nomenclature. The elements of a C++ `enum`, to which the HASE `ENUM` construct is mapped, can only contain alphanumeric characters and the first character must always be a letter. This impacts on the way the function field of an instruction can be formulated. In the MU5 model, for example, the function '`NB =`' is represented as '`NBld`'. Choosing ways to represent the instruction formats, both within the simulation code and on-screen, were among the most significant design decisions in the creation of these models.

Atlas

The Ferranti Atlas computer resulted from the fourth major computer design and implementation project carried out at the University of Manchester. The first Atlas was inaugurated at the University in December 1962. Designed by Professor Tom Kilburn and a joint University/Ferranti team, it incorporated a number of novel features, the most influential of which was Virtual Memory. IEEE Milestone Plaques recognising this achievement and the "Manchester University 'Baby' Computer and its Derivatives" were presented to the University on 21st June 2022.

Atlas had a 48-bit instruction word made up of a 10-bit function code, two 7-bit index addresses (Ba and Bm) and a 24-bit store address (Figure 2). Functions that operated on the Accumulator (A-codes) were thus one-address with double B-modification. B-register 0 always returned a value of 0, thus allowing for singly modified or unmodified accesses. The most significant bit of the address field distinguished between user addresses and system addresses, while the three least significant bits were used to address one of eight 6-bit characters within a store word.

The B-codes were executed by the B-Arithmetic (add/subtract and logical) Unit and operated on one of the 128 24-bit B-registers specified by the Ba field in the instruction, Bm being used to specify a single modifier. The B-store was made up of 120 words of core store together with eight special purpose flip-flop registers. These included the floating-point accumulator exponent, for example, and the three control registers (program counters) used for user program, extracode and interrupt control. Including these three control registers as B-registers avoided the need for separate control transfer (branch) functions. Extracodes were a set of functions which gave access to some 250 built-in subroutines held in the read-only Fixed Store.

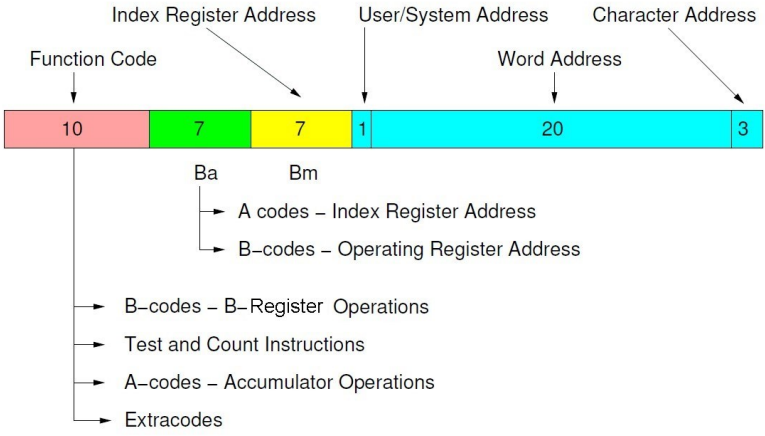


Figure 2 : Atlas Instruction Format

User addresses in Atlas referred to a 1M word virtual address space, and were translated into real addresses through a set of page registers. The real store consisted of a total of 112K words of core and drum storage, combined together through the paging mechanism so as to appear to the user as a one-level store. System addresses referred directly to the Fixed Store, the Subsidiary Store (used as working space by the extracodes and operating system), or the V-store. The latter contained the various registers needed to control the tape decks, input/output devices, paging mechanism, etc., thus avoiding the need for special functions for this purpose.

A screen image of the HASE Atlas model is shown in Figure 1. Because integers, reals and instructions cannot be mixed together in a single array, the model takes advantage of the Atlas paging system by modelling the Drum Store as a set of pages and the Core Store as a set of blocks, each made up of 512 words, as in Atlas, but with different blocks/pages containing different types of element. This is unrealistic, of course, but was felt to be an acceptable compromise given the intended use of the model.

At the start of a simulation the program and its data are contained in the Drum Store while the Core Store is empty (i.e. contains zeroes). The program code is in page 0 of the Drum, fixed-point integers are in page 2 and floating-point reals in page 3. In the Core Store, Block 0 is modelled as an instruction array, Block 1 as an integer array and Block 2 as a floating-point array. These arrays are themselves loaded from files such as **`CORE STORE.block0.mem`** and **`DRUM STORE.page0.mem`** when the model is loaded into HASE. The transfers from the Drum Store to the Core Store take place instantaneously "behind the scenes", rather than being simulated as word-by-word transfers through the Distributer, since this would be extremely tedious to watch in the playback. The Fixed Store and Subsidiary Store are also included as entities in the model, although the Fixed Store is not actually used.

The Atlas instruction set is modelled using the HASE INSTR construct. INSTR allows functions to be grouped together both for decoding purposes and to allow each group to be associated with an appropriate addressing STRUCT. In Atlas, the addressing STRUCT contains three integer values, Ba, Bm and address, and is common to all instructions:

HASE maps the functions within a function group to a C++ enum, so all functions must start with a letter. The INSTR construct was created during the development of a HASE DLX model and because the DLX instruction set is represented using an assembler code nomenclature, this constraint was not an issue. Functions in the Atlas instruction set are represented numerically, however, so three possible solutions were considered to the quandary that this presented: (a) create an assembler code style representation of each function; (b) simply represent instructions using a STRUCT containing four integers (thus foregoing the built-in decoding facility); (c) prefix each function number by 'A' for Accumulator functions, 'B' for B register functions and 'E' for Extracodes. Option (c) was chosen, so the Accumulator group of functions, for example, is represented within the INSTR definition as

(ACC(A314,A315,A320,A321,A322,A346,A356,A362,A363,A374)

This offers some limited insight into the purpose of each instruction but it was felt that an on-screen method of conveying the full meaning of each instruction to the user would be valuable. This was achieved by coding, in the *global_fns* file, a **`get_text`** function that returns a text string for each function, e.g. '**`b=b+s`**' for function B104. This is then displayed on the Control icon (see Figure 1).

Three versions of the model are available from the website, each containing a different machine code program. The first is designed to demonstrate the operation of each of the instructions implemented in the model, while the second demonstrates the operation of matrix multiplication, one of the most frequently used algorithms in supercomputers. The third is an adaptation of a program that finds values for the lengths of the sides of Pythagorean right triangles. This program uses three Extracodes, B multiply and two print instructions. In the

model, the B multiply instruction is executed directly in the B-Arithmetic Unit while the print instructions are implemented by the Control Unit and write values into the Subsidiary Store. These values are printed at the end of the simulation run in the output pane of the HASE GUI. An extension of the model could be to implement at least some of the Extracodes using code held in the Fixed Store, as in the real Atlas.

MU5

MU5 was the fifth computer system to be designed and built at the University of Manchester, in succession to Atlas. MU5 introduced a number of new ideas, particularly in terms of its instruction set, which was designed with high-level language compilation in mind, and novel uses for associative stores. The technical aspects of the project have been documented in numerous publications. MU5 became operational in the mid-1970s and ran as a service machine in the Department of Computer Science until 1982. Many of the design ideas developed in MU5 were used in the ICL 2900 series.

Type of Function	
Org	Organisational
B	Index arithmetic
STS	Store-to-Store
Acc	Accumulator

Kind of Operand	
lit	Literal (n)
V32	32-bit Variable (name n)
V64	64-bit Variable (name n)
S0	Data Structure (name n) (unmodified access)
SB	Data Structure (name n) (modified access)
stack	64-bit operand at SF SF = SF - 2

Base	
Z	0
NB	Name Base
XNB	Extra Name Base
SF	Stack Front
D	Use current D

Figure 3 : MU5 Simulation Model Instruction Set

It was clear at the outset of creating the MU5 model that it would not be possible to use the HASE INSTR construct to accurately represent the MU5 instruction set. The instruction set used in the model (Figure 3) therefore follows the spirit of the original, rather than the detail, in particular because the instruction format and length have to be fixed in HASE, rather than both being variable as they were in MU5. So instructions in the model are represented by a `STRUCT` containing 4 enumerated types, one each for the type of function, the function itself, the kind of operand and the base register, together with an integer used as a name or literal. Using a `STRUCT` of this complexity had not previously been attempted in HASE and modifications to the EDL parser were needed in order to process it.

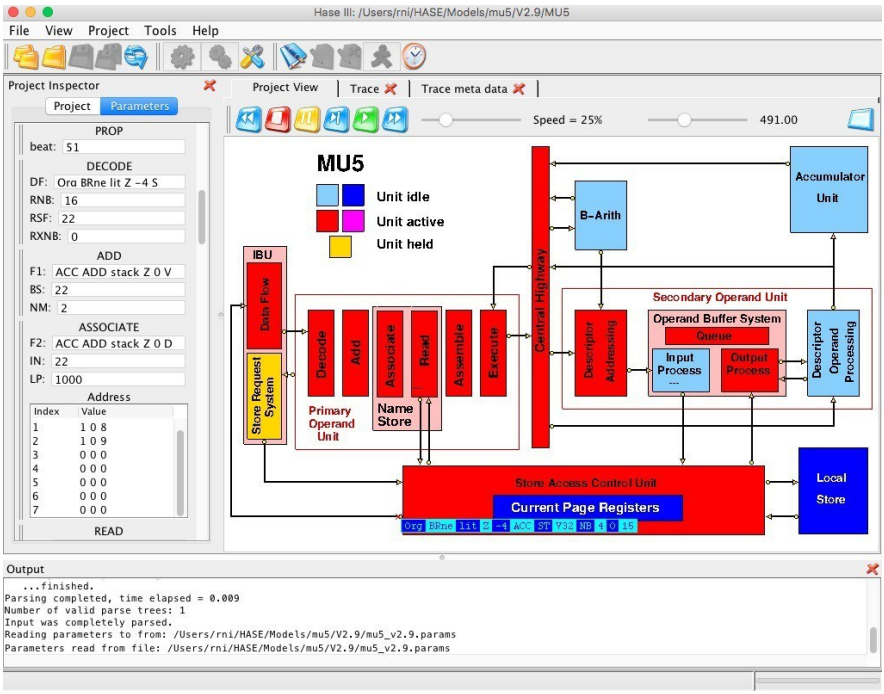


Figure 4 : The HASE MU5 Model

Figure 4 shows the HASE MU5 model, the design of which closely follows that of MU5 itself. At the start of a simulation, the address contained in the Control Register (program counter), CO, in the Execute stage of the Primary Operand Unit pipeline (PROP), is sent to the Store Request System within the Instruction Buffer Unit (IBU). The IBU sends this address to the Store Access Control Unit (SAC), which contains a set of Current Page Registers (CPRs). MU5 itself had 32 CPRs, but the HASE model has just four, preloaded such that the first page of Segment 0 (the Name Segment) is mapped to Block 0 of the Local Store, Segment 1 is used for instructions and is mapped to Block 1, while Segment 2 is used for array elements and is mapped to Block 2. Segment 3, mapped to Block 3, is used to contain strings of bytes. Implementing software to manipulate the CPRs is beyond the scope of this model.

SAC sends the translated address to the Local Store, which returns the two instructions contained in the addressed memory word back to SAC, which itself sends these instructions to the IBU. The IBU enters the instructions into its buffer registers (organised as a drop-down stack) and sends each instruction in turn to PROP. PROP processes instructions in a six-stage pipeline, at the end of which it has accessed and prepared the appropriate primary operand associated with each instruction and then, depending on the instruction type, either forwards the instruction to the B Arithmetic Unit or the Secondary Operand Unit,

or, in the case of an organisational function, executes the instruction itself. Instructions sent to the Secondary Operand Unit are those destined for the Accumulator Unit or others that require a secondary operand accessed via the descriptor mechanism. In the hardware of MU5, various extra function bits were added to instruction registers in the pipeline in order to control the different activities that could occur at each stage. In order to replicate this mechanism in the model and to represent the bits in a visually meaningful way, each instruction register includes an additional character, the T field shown in Figure 3.

Three versions of the model are available from the website, each containing a different program. Version 1 contains a program that demonstrates the operation of the MU5 Name Store, Version 2 contains a program that executes a scalar product while Version 3 contains a program that demonstrates the use of the string processing instructions. This program performs three tasks. The first creates the composite string "**Hello world**" from individual strings containing its component parts. The second finds the first non-zero digit in the digit string 00083576 and places the ASCII representation of its value in the B register. The third searches for the word '**light**' in the sentence '**Let there be light.**'; the B register then indicates its position within the sentence.

For visualisation purposes, the strings are held in character format in their own block of the Local Store, with each word containing eight characters. These characters are processed using the same simulated hardware as that used for integer values, however, so to get round type checking constraints, the characters are converted to their ASCII integer values and packed into a pair of 32-bit integer words whenever they are read out of memory and, correspondingly, returning integer words are unpacked during a write-to-store operation. Similarly, in the Descriptor Operand Processing Unit, each character is manipulated as an integer value so an extra register is used to display the actual characters.

CDC 6600

The CDC 6600 was first demonstrated by the Control Data Corporation in 1964. The design team was led by Seymour Cray, who went on to design the CDC 7600 and later, after he left CDC to form his own company, the Cray-1. The 6600 was designed to solve problems substantially beyond contemporary computer capability. It achieved its performance by the use of parallel functional units, a small set of Scratch Pad registers, a three-address format (Figure 5) that allowed successive instructions to refer to totally independent input and result operands, instruction buffering and by off-loading peripheral handling to separate peripheral processors. Particularly interesting to students of computer architecture is the Scoreboard system used to control the operation of the parallel functional units..

Figure 6 shows the HASE simulation model of the CDC 6600 during execution of the first of the two versions of the model available from the website. This version

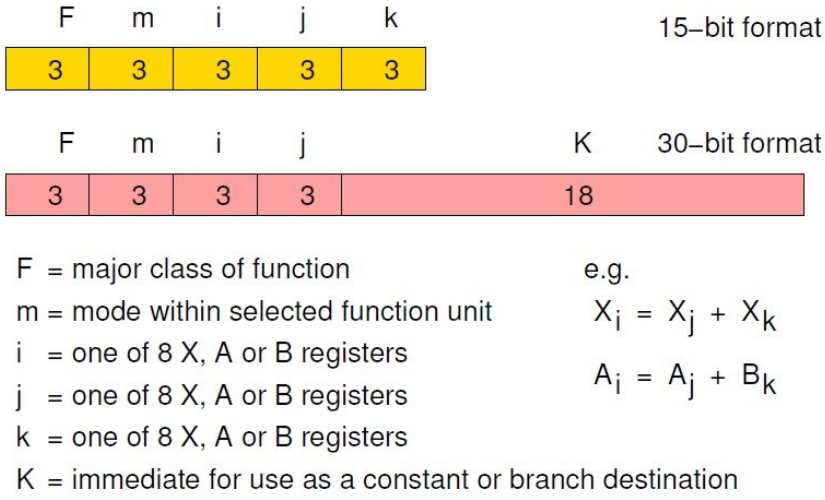


Figure 5 : CDC 6600 instruction format

contains a program that demonstrates the operation of many of the CDC 6600 instructions implemented in the model. In particular, it includes sequences of instructions that show how the Scoreboard deals with instruction dependencies. The second version contains a matrix multiplication program that shows how the instruction stack works and how instruction unrolling of the scalar (dot) product calculation (the inner loop of matrix multiplication) is used to maximise register use.

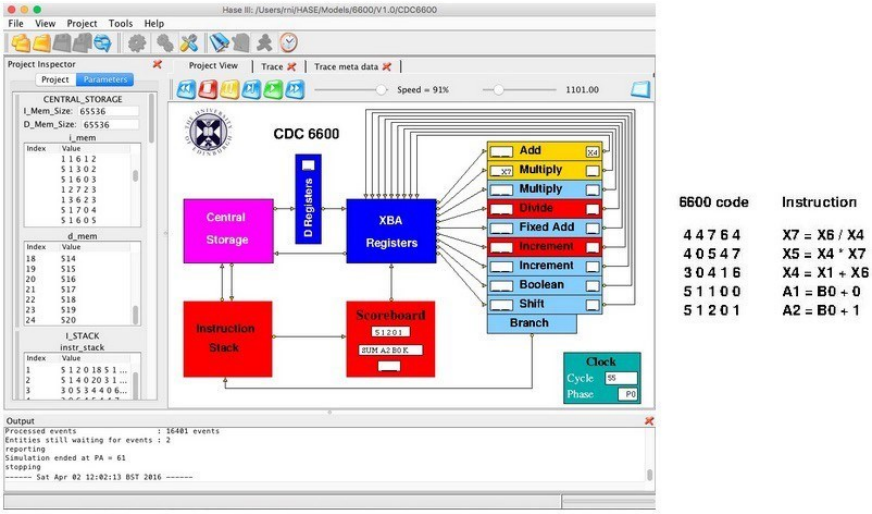


Figure 6 : CDC 6600 Simulation Model

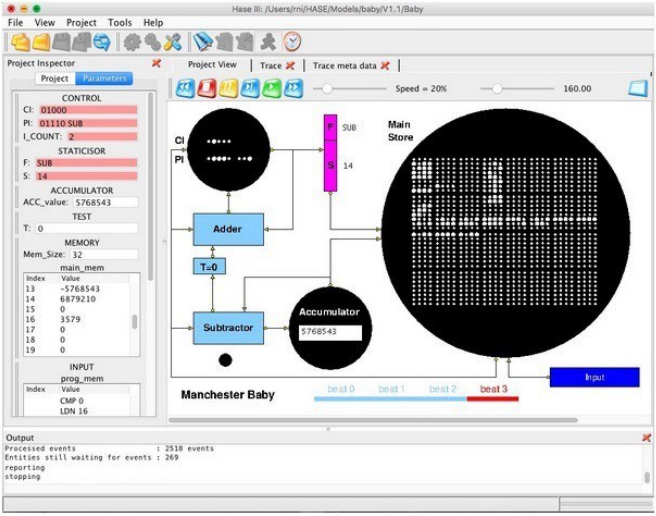
In principle it ought to have been possible to represent the 6600 instruction set in the model using the INSTR construct but in practice there is no regular relationship between groups of functions and the register sets, so most instructions have to be decoded, and their registers selected, individually. Instructions are therefore represented using a STRUCT containing 5 integers and instructions are decoded in the Scoreboard simulation code using a switch statement. The code for each case in the switch selects the required registers and also contains text to be displayed in the Scoreboard icon showing the assembly code representation of each instruction.

On screen, active entities (other than memories) are coloured red, inactive entities are shown in light blue, while entities that are held up waiting for a packet from some other entity are shown in yellow. In the functional units in Figure 6 the functional unit colours appear correspondingly as dark grey, light grey and white. Also shown on the right in Figure 6 are the instructions in the sequence currently being processed. The first instruction is being executed in the Divide Unit. The second instruction has been issued to the first Multiply Unit but cannot start because it is waiting for the result from the Divide Unit to be returned to X7, as shown by the entry in the window at the left-hand end of the Multiply Unit. This is known as a "second-order conflict" in 6600 terminology, but would nowadays be called a "read-after-write dependency". The third instruction has been issued to the Add Unit, but cannot start because of a "third-order conflict" (a "write-after-read dependency") on X4, as shown in the window at the right-hand end of the Add Unit, i.e. it is ready to write its result to X4, but the current value in X4 has not been sent to the Multiply Unit. The design of the 6600 was such that register values are sent to a functional unit only when both are ready, i.e. not waiting for a result from a previous instruction; here X7 is not

yet ready. The next instruction does not depend on any previous results, so has been issued to the first Increment Unit and is being executed. The Scoreboard is displaying the last instruction, which is yet to be issued. The window at the bottom of the Scoreboard is used when there is a "first-order conflict", either because the instruction about to be issued requires a functional unit that is already busy or because its destination register is already waiting for a result from another unit (a "write-after-write dependency").

The Manchester 'Baby'

On 21st June 1948 the University of Manchester Small Scale Experimental Machine, affectionally known as the 'Baby', became the first computer to execute a program (Kilburn's Highest Factor program) stored in addressable read-write electronic memory. This memory worked by storing charge on the screen of a cathode ray tube, now known as a Williams-Kilburn Tube. Like the Baby itself, the main emphasis of the HASE Baby model (Figure 7) is to demonstrate the operation of the Williams-Kilburn Tube Main Store. This requires its icon to be large enough to clearly display the Baby's 32 lines of 32 bits each. To allow for this, the Accumulator and CI/PI icons are much smaller. In the case of the



Line	Instruction	Action
1	LDN 24	load $-b_1$
2	STO 26	store $-b_1$
3	LDN 26	load $+b_1$
4	STO 27	store $+b_1$
5	LDN 23	load a
6	SUB 27	trial subtraction
7	CMP 0	is difference negative?
8	JRP 20	no, jump back 2 lines
9	SUB 26	overshot, add back b_n
10	STO 25	store $+r_n$
11	LDN 25	load $-r_n$
12	CMP 0	is remainder zero?
13	STP 0	yes, stop
14	LDN 26	no, load $+b_n$
15	SUB 21	form $b_{n+1} = b_n - 1$
16	STO 27	store b_{n+1}
17	LDN 27	load $-b_{n+1}$
18	STO 26	store $-b_{n+1}$
19	JMP 22	jump to line 5

Line	initial	current	final	Comment
20	-3			line 8 operand
21	1			line 21 operand
22	4			line 19 operand
23	-a			trial number
24	b_1			initial trial factor
25	0	r_n	r_N (=0)	remainder
26	0	$-b_n$	$-b_N$	
27	0	b_n	b_N	

Figure 7 : The HASE Baby Model

Accumulator a further compromise is that the value is displayed as an integer rather than a pattern of 32 dots and in the case of the CI/PI icon, only the relevant bits of the CI and PI words are displayed, i.e. 5 bits and 8 bits respectively. The PI bits are only visible whilst PI is in use.

At the start of a simulation the program and its data are contained in two separate files held in the Input entity. In the current version of the model, these files can contain the code and data for four programs (there are currently three

in the model). A parameter of the Input entity allows the user to select the program to be run. During the first clock period of the simulation the relevant sections of these two files are transferred into the Main Store entity. Entries from the program file are converted from the human-readable form in the input file into Baby format and transferred to the Main Store starting at location 1 (because the first action in the Baby was to increment the value (initially 0) in the CI register). The remaining Main Store locations are then filled with values from the input data file. Each word in Main Store is then displayed on the screen in focus/defocus format, with a small (focussed) dot representing a 0 and a large (defocussed) dot representing a 1, and with the least significant digit at the left-hand end.

Reading a line in a Williams-Kilburn Tube involved writing a 1 to each bit location in the line. If the location contained a 0, there was a redistribution of charge on the screen and this induced a voltage in a pick-up plate placed close to the screen. If the location contained a 1, there was no redistribution of charge, so no voltage was induced. In the HASE model a line being read from the Main Store is shown changing to all 1s and then changing back to its original value, though this is purely a feature of the display; the Main Store array remains unaltered and is simply read by an assignment statement.

A further difference between the model and the Baby itself concerns the way in which the serial nature of the Baby is visualised. Watching 32 bits transfer between entities at a rate of one per clock period would be extremely tedious, so words and addresses are each transferred in a single HASE clock period with their integer values represented by strings of 0's and 1's. The time taken to read a full word from the Main Store, plus the fly-back time, constituted a 'beat' of the machine. Four beats were required to execute an instruction. In the model these are:

add +1 or +2 to CI, copy CI to staticisor
read Main Store, send instruction to PI
copy PI to staticisor, fetch operand from Main Store
execute the present instruction

The timing bar at the bottom of the Project View pane shows which beat is active as each instruction is executed. The black dot below the Subtractor represents the neon lamp that lights up when a Stop instruction is executed.

Of the three programs contained in the model, Program 1 demonstrates the operation of all the Baby instructions, while Program 3 displays a simple message on the Main Store icon. Program 2, for which the code (in Chris Burton's modern mnemonic version) and data are shown on the right in Figure 7, is the 18th July 1948 version of the Highest Factor program, as described by Geoff Tootill, along with some of his explanatory comments. The number under investigation, a, is set to `4537` (`=13x349`) in the input data file, this being one of the numbers used in the first tests of the Baby on 21st June 1948. The initial trial

factor, b1, for that test was 4537 but in the model it is set to 360 to avoid a very long simulation run. At the end of a simulation run, the highest factor of 4537, 349, is in Line 27 of the Main Store. Of course users of the model can set the input values to numbers of their own choice.

Conclusion

Simulation models of historical computers that provide visual demonstrations showing how their various architectural features operated offer effective alternatives to preserving, in working order, what were in many cases very large pieces of equipment. Having simplified models in an application such as HASE allows anyone interested to down download a model and observe it in action.