The University of Newcastle
School of Information and Physical Sciences
COMP3290 Compiler Design
Semester 2, 2025

The SM25 Architecture

Introduction:

During this semester, the projects in COMP3290 will focus on writing a complete compiler for the language CD25, for the machine known as the SM25 (Stack Machine 2025), which is a small hand-held device designed to work in extreme conditions, such as in bushfires, caving expeditions, radioactive areas, or even on an expedition to Mars. The architecture of the SM25 has been developed from the 1970's era Burroughs B6700 architecture.

Part of this project is to evaluate just how useful a device such as the SM25 will be, by evaluating the complexity and utility of programs that can be executed on it. You will execute your compiled CD25 programs on a Simulator for this architecture.

Normally the SM25 will download programs by shielded coaxial cable and will accept data from a specialized keypad and display results simultaneously on a toughened screen, audibly using Morse code, and on a special Braille touchpad. For your experiments on the simulator, the input and output devices, and the program download device will be simple text files.

The SM25 Simulator:

The simulator uses a Graphical User Interface. It has 2 buttons for the initialization of the runtime system, ready for a program to execute:

1. ***"Load File"*** which reads an initial system state from a code module file, and
2. ***"Reset"*** which re-initializes the currently loaded code module.

It also has 4 buttons to execute the program:

1. "***Run to Halt"***- execute the program until a HALT instruction is executed,
2. ***"One Instruction"*** - execute one instruction and update the display to reflect the system's current state, and
3. ***"Ten Instructions"*** - similar to 10 lots of "One Instruction" except that it will stop on a HALT instruction if executed before the 10 are reached.
4. ***"100 Instructions"*** - similar to 100 lots of "One Instruction" except that it will stop on a HALT instruction if executed before the 100 are reached.

For more information on the simulator, please consult the **SM25 Simulator User manual**.

Stack Machine Structure:

The SM25 machine has a number of interesting features, including on chip, register memory. The addressing structure allows this to be up to 4Gb, but the test implementation uses just 64 Kbytes, of which about 54 Kbytes are available to run user programs, the rest being given over to the Operating System (SMOS25).

The basic architecture followed for the SM25 is that of a "*Stack Machine*". This means that most instructions do not carry any operand information within the instruction itself, but operate on a stack of registers. Before an instruction can be executed, the programmer (or compiler writer) must ensure that the necessary data items (*operands*) are already available at the top positions of the stack, ready for the instruction to be performed. Also, if the instruction produces a result, then this will be stored back onto the stack in place of the operands that will have now been popped.

There are only 2 instructions that refer directly to memory, **LA** (Load Address) and **LV** (Load Value). Another small set of instructions give side effects, which alter memory somewhere other than the top of the stack.

Memory Protection Tags

The memory is designed with hardware protection, each memory word of 64 bits carrying a 4-bit tag, which shows its type. Memory is, therefore, actually 68 bits wide. Once a program creates a piece of data, it is not allowed to use that piece of data in ways other than those legal for that type. So an address cannot have an integer added to it, and an integer or a floating point data item cannot be used as a memory address or return target for a procedure call. An instruction such as **MUL** (for Multiply) will only work if

1. both its operands are integer quantities, and will produce an integer quantity as its result, or,
2. both its operands are floating point quantities, producing a floating point result.
3. one of its operands is floating point, and the other integer, whereupon the integer is promoted to floating point and a floating point result produced.

Note that the tagged memory means that the one instruction can be used for both integer addition operations and floating point addition operations. The same is true for most other arithmetic and relational instructions.

Instructions can only be used by the fetch-execute cycle of the machine, and other data cannot be used as instructions. On top of this, the SM25 knows where instructions are stored and where other data is stored and will not fetch instructions from other areas of memory. Also, this instruction area cannot be overwritten. Constant data items are also protected, by storing them within the "*instruction area*" of the SM25 (but carrying their correct tag). This means that they, also, cannot be overwritten.

One special tag is **UNDF** for undefined. Data items, which have been allocated but not yet initialized, are given this tag, so that un-initialized data can't be used in a calculation.

A complete list of memory tags is:

- **UNDF** for *undefined*,
- **INST** for *instructions*, can only be processed by the SM25 fetch-execute cycle,
- **INTG** for *integer*,
- **FLOT** for *floating point* (real) values,
- **BOOL** for *boolean* (the result of a relational or logical operation),
- **STRG** for *string* constants (also stored in the instruction area),
- **ADDR** for the setting up the *addresses* of targets for storing data values or branch instructions,
- **DESC** for array *descriptor* (see below), and
- **MSCW** for "*mark stack control word*" (also see below), which is used to establish a call frame for a procedure or function.

To allow a compiler to forcibly alter data types, there are 3 special instructions that will directly alter the memory tag of certain types of operand on the stack.

Arrays:

Arrays have specific hardware support via an array descriptor and access to elements of an array carries hardware support for "*index out of bounds checking*", as well as the regular type checking mentioned above. Array elements (the array body) are initially allocated using the **UNDF** (*undefined*) tag, so that they must also be initialized before they are used. Note that an array descriptor does not specify the type of values that are to be stored into the array itself, so an array may contain *Integer* or *Float* or *Boolean* values (or even a mixture of the three).

Support for Sub-Programs:

SM25 also contains hardware support for procedure calling, to especially support languages such as C (or C++ or Java, but there is *no obvious support for OO programming*), by means of a base register which acts as a frame pointer for a procedure call. Procedure/Function Calling instructions and Procedure/Function Return instructions provide sufficient support for recursion and single-block structured programming languages such as C.

Languages such as Pascal, which allow functions to be defined within other functions are not specifically supported by this architecture.

Base Registers and Stack Pointer:

A main part of the architecture is the set of ***three Base Registers*** (**b0**, **b1** and **b2**).

- **b0** points to the instruction space;
- **b1** to the main program space, and;
- **b2** to the call-frame of the most recently called procedure/function.

The ***Entry Point*** (**ep**) is the first word of the instruction space (ie: always 0 bytes beyond **b0**), the beginning of the instructions themselves, and then the data constants area. The SM25 also maintains the program's ***Instruction Limit*** (**il**), and a ***Stack Pointer*** (**sp**) pointing to the latest item pushed onto the stack.

This means that at any time, the execution environment for the program consists of the constant data, the global data space, and the currently executing procedure's environment.

The constant area is limited by **il** and **b1,** and accessed via **b0**, except for the single data allocation word which **b0** points to directly.

The global data space is pointed to by **b1** and limited by the call frame of the first procedure called (or the stack pointer (**sp**) if no procedure is currently active).

The currently executing procedure's environment (the call frame, locals and temporary values) is pointed to by **b2**, and its limit is given by the value of the stack pointer (**sp**).

The Program Counter:

This next instruction byte to be fetched is pointed to by the ***Program Counter*** (**pc**), and this is automatically incremented when an instruction is fetched. Branch, Subprogram Call, and Return instructions involve storing a new value back into the **pc** register. The **pc**, **ep**, **il, b0**, and **b1** registers (*along with the special memory tag for instructions*) work together to protect the program from fetching data as instructions, and from storing data into the instruction space, or overwriting constant data.

Memory:

The SM25 is a byte addressable computer, and instructions can be fetched one byte at a time, but the *integer*/*float*/*string constant* areas, and the stack itself all use words of 64-bits allocated on word boundaries, so all the base registers, the entry point register and the stack pointer will always be *a multiple of 8*. The SM25 is also a *"big-endian"* machine with instruction bytes being fetched from the most significant bits first, that is, the word beginning at address 1024 consists of high-order byte 1024 through to low-order byte 1031.

The Instruction Set

As previously stated, most instructions have *implied operand positions* and *implied result target positions* at the top of the stack. For the following *arithmetic* means a data item that has either the **INTG** or **FLOT** tag. A complete instruction list is as follows:

Opcode	Instruction	Operands	Result
00	**HALT**	none	Stop execution
01	**NO-OP**	none	Do nothing
02	**TRAP**	none	Stop Execution – Abort
03	**ZERO**	none	Push the INTG zero onto the stack
04	**FALSE**	none	Push the BOOL false onto the stack
05	**TRUE**	none	Push the BOOL true onto the stack

Opcode	Instruction	Operands	Result
07	**TYPE**	1 x arithmetic	swap types INTG >> FLOT, FLOT>>INTG
08	**ITYPE**	1 x arithmetic	set type for TOS to INTG
09	**FTYPE**	1 x arithmetic	set type for TOS to FLOT
11	**ADD**	2 x arithmetic	add them, push the result
12	**SUB**	2 x arithmetic	subtract first popped from second, push
13	**MUL**	2 x arithmetic	multiply them, push the result
14	**DIV**	2 x arithmetic	divide second popped by first, push
15	**REM**	2 x Integer	second popped MOD first, push
16	**POW**	1 x arithmetic	
		1 x Integer	raise second popped to power of first, push
17	**CHS**	1 x arithmetic	push negative of operand popped
18	**ABS**	1 x arithmetic	push absolute value of operand popped
21	**GT**	1 x arithmetic	if TOS > 0, push true else false
22	**GE**	1 x arithmetic	if TOS >= 0, push true else false
23	**LT**	1 x arithmetic	if TOS < 0, push true else false
24	**LE**	1 x arithmetic	if TOS <= 0, push true else false
25	**EQ**	1 x Integer	if TOS == 0, push true else false
		…….. 1 x Float	if | TOS | < 0.000001, push true else false
26	**NE**	1 x Integer	if TOS != 0, push true else false
		…….. 1 x Float	if | TOS | > 0.000001, push true else false
31	**AND**	2 x Boolean	if (popped) (b1 & b2), push true else false
32	**OR**	2 x Boolean	if (b1 | b2), push true else false
33	**XOR**	2 x Boolean	if (b1 exclusive-or b2), push true else false
34	**NOT**	1 x Boolean	push logical negation of popped item
35	**BT**	1 bool. 1 addr	if boolean is true then place address into pc
36	**BF**	1 bool, 1 addr	if boolean is false then place address into pc

37	**BR**	1 x Address	place address into pc
40	**L**	1 x Address	pop address, push the value at this address
41	**LB**	1 x Instr Byte	push 8-bits, sign extended to 64, as INTG
42	**LH**	2 x Instr Byte	push 16-bits, sign extended to 64, as INTG
43	**ST**	1 arith/bool, 1 addr	store value (first) popped at address popped
51	**STEP**	none	step the SP one word (tagged UNDF)
52	**ALLOC**	1 x Integer	step the sp by this many (popped) words
53	**ARRAY**	1 int, 1 addr	construct descriptor and step sp by size
54	**INDEX**	1 int, 1 desc	construct address of elt (int) in array (addr)
55	**SIZE**	1 descriptor	pop descriptor, extract & push array size
56	**DUP**	none	push a duplicate of the top item on the stack

Opcode	**Instruction**	**Operands**	**Result**
60	**READF**	none	input floating pt value, push to stack
61	**READI**	none	input integer value, push to stack
62	**VALPR**	1 x arithmetic	print a space and then the popped value
63	**STRPR**	1 x Address	print string const at popped address
64	**CHRPR**	1 x Address	print character const at popped address
65	**NEWLN**	none	terminate the current line of output
66	**SPACE**	none	print a single space character
70	**RVAL**	1 x arith/bool	pop value, store in function return position
71	**RETN**	none	pop the proc environment, return to caller
72	**JS2**	1 int, 1 addr	construct call frame, branch to proc/fn

Special Memory Referencing Instructions

80 These form a family of **LV** (*Load Value*) instructions. 0/1/2 gives the base
81 register number. The instruction code is followed by a *4-byte offset*.
82 ***Base Reg*** plus ***offset*** give an ***address***, the value at this address is pushed.

90 These form a family of **LA** (*Load Address*) instructions. 0/1/2 gives the
91 base register number. The instruction code is followed by a *4-byte offset.*
92 ***Base Reg*** plus ***offset*** give an ***address***, and this address is pushed.

Some Extra Detail

Non-commutative arithmetic instructions:
perform 1st-item-pushed operator 2nd-item-pushed,
eg. *x – y*, has the code structure: *load value of x, load value of y, subtract*.

ARRAY: An array descriptor is a single 64-bit word, with the high order 32-bits holding the array's size and the lower order 32-bits holding the start address of the block of memory to be allocated for the array (ie the next value of the stack pointer). Detailed semantics of **ARRAY** are:

- Pop the array size operand,
- Pop the address operand as the place where the descriptor is to be stored,
- Form an array descriptor from the array size and the value of "sp+8" and store it at the descriptor address
- Step the stack pointer by the number of words in the array size (sp += 8*size).

INDEX: The first item pushed is the value (not the address) of the array's descriptor, the second is the (integer) element number (in the range 0 up to size-1). **INDEX** will check the element number for "out of bounds", and then create the address of that element using the start address value in the descriptor and the element number, and then push that address.

JS2: The only difference between a procedure and a function in SM25 is that a function is deemed to have had a place allocated (just above the call frame) to hold the return value of the function. It is also assumed that all parameters have been pushed before the JS2 instruction is issued.

The two operands to **JS2** are the number of parameters (pushed first) and entry point of the subprogram (pushed second). **JS2** creates a Mark-Stack-Control-Word and pushes this. This **MSCW** is made up of the saved value of the b2 register (the caller's Frame Pointer – high-order 32-bits) and the return address of the caller (low-order 32-bits). The **b2** register is now set to point at this **MSCW** and the number of parameters (operand) is then pushed back onto the stack.

Within the subprogram, parameters will be accessed by way of negative offsets from the **b2** Frame Pointer, **b2** itself will point to the **MSCW**, and offset +8 from **b2** will give the number of parameters. If the subprogram is a function, then it must issue the **RVAL** instruction before returning (**RETN**) from the subprogram.

RETN: The return instruction removes the entire call frame, including the parameters and stores the return address back into the pc. The old (caller's saved) value of **b2** is restored, re-establishing the caller's environment. If the subprogram was a function then the return value will be left at the top of the stack.

RVAL: The set-return-value instruction relies on the call frame structure and return value space allocation mentioned above. The return value is popped from the stack and stored at the word, which is immediately above the function call frame. If the sub-program is a function then this instruction must be executed at least once by the function before it returns, otherwise the caller may be left with an allocated function return value, which is still undefined (tag **UNDF**).

Memory Addressing and Offsets:

All memory is addressed by giving a Base Reg number and an offset, which are added to give the byte address of the data item or instruction required – *provided it is within the legal range for that form of addressing*. The offset is stored in 8 bytes following the relevant opcode (Load Address or Load Value).

Parameters are stored above the subprogram's MSCW word (pointed to by **b2**) and so are referenced by means of negative offsets from **b2**. All offsets are stored in 4 bytes and so negative offsets are stored in *2's complement* form with the high order byte first followed by the low-order byte. So an offset of *-8* will consist of the bytes *255*, *255*, *255* and *248*, because *255*256**3+255*256**2+255*256+248* is the 32 bit *2's complement* value that represents *-8*.

Invalid Items:

If at any time the SM25 finds data or operands or target addresses which are inconsistent with the type and function of the information being used, then SM25 will issue at trap to its Operating System (*a Fatal SM25 Exception within our simulator*).

Once a fatal exception has been thrown, the simulator will not abort but will wait for either the issue of a ***"Reset"*** command, so the same program can be run again – perhaps with different input and output files, or for a new Module Filename (*perhaps with new input and output files*) to be entered and ***"Load File"*** chosen.

v1.01: *2025-07-15*