APPENDIX A

The JASP Toolkit

A.1 Introducing JASP

The JASP toolkit is based around the design of a simple processor named JASP - *Just Another Simulated Processor*. A diagram of this processor is shown in figure A.1.

The original JASP design was by William Henderson of the School of Informatics, Northumbria University.

JASP consists of a small set of registers, a microcoded control unit and an ALU. This processor is connected to a memory to form a rudimentary computer system.

JASP is deliberately meant to be simple technology - containing the simplest elements from a microcoded processor. JASP is designed as an educational tool to demonstrate fundamental concepts in a generic way, so allowing students to gain an understanding of them prior to transferring to ‘real world’ processors.

Within the toolkit we have two simulations of the JASP processor, the main one being JASPer (*Just Another Simulated Processor emulator*), the second being Aspen (*Another Simulated Processor emulation*). The toolkit also contains an assortment of useful tools to aid the use of the processor simulators.

The set of tools includes:

- ► *JASPer* - the main simulated processor, JASPer will run on Windows 95 and above;
- ► *Aspen* - a command-line version of JASPer that can be used with DOS or Linux;
- ► *The JASP cross-assembler* - a cross-assembler, written in Perl, that assembles programs for the JASP architecture. The assembler should work on any platform with a Perl installation, but it has been tested under DOS and Linux only;

- *The JASP C−− cross-compiler* - a cross-compiler for the JASP architecture. It can be used with DOS or Linux;
- The basic and advanced JASP instruction sets;
- Two software libraries, for use with each instruction set.

A reference to the functionality of the JASP processor is given below, followed by descriptions of each tool in the toolkit.

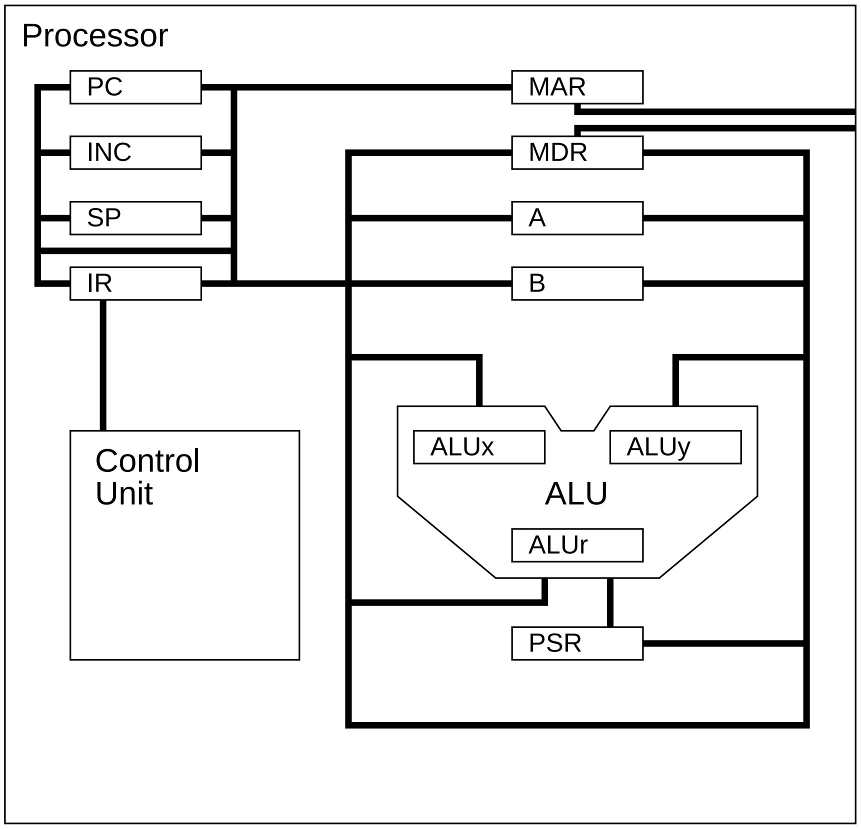


Figure A.1 ***The JASP processor***

A.1.1 Obtaining The JASP Toolkit

All the tools within the JASP toolkit are copyright Mark Burrell, except for the C−− cross-compiler which is copyright David Harrison.

The authors of this toolkit, and the distributors, cannot accept responsibility for any disruption, damage and/or loss to your data or computer system that may occur while using this package. This is free software and comes with no warranty.

To obtain a copy of the JASP toolkit go to this website:

- `http://www.palgrave.com/science/computing/burrell/`

Additionally, a copy of the JASP toolkit is available on the accompanying CD.

While the authors hold copyright on all JASP executables and documentation, it may be freely distributed at no cost (excluding minimal expenses), providing that it is distributed intact, and not subsumed into any other work.

A.1.2 Installing The JASP Toolkit

The JASP toolkit has various tools, some of which work in Microsoft Windows, while others work on Linux systems. The installation instructions for each environment are given below.

Refer to the `readme.txt` file within the toolkit for a detailed description of each file, and how it can be used.

Installing On Windows

The JASP toolkit is distributed as a zipped archive, and you need to unzip this archive into a directory on your computer. None of the JASP tools make use of the Windows registry, and so no further installation is necessary, apart from you may set up any desktop icons or menu options as you prefer.

You can additionally set up an environment variable called `JASP`, and update your path as listed below - but this is only really required if you intend to make use of the JASP assembler or Aspen.

If you do wish to update your path and use the `JASP` environment variable then place the following lines in your `autoexec.bat` file (this assumes you have installed the JASP toolkit at the location `c:\jasp\`):

```
set path=%path%;c:\jasp\

set JASP=c:\jasp\
```

Installing on Linux

Once again, unzip the archive into a directory on your computer - something like `~/jasp`.

As all the tools for Linux are command-line driven, it makes sense to set up the path and the `JASP` environment. You may need to do this differently, depending on your flavour of Linux, but on my system, using the `bash` shell, all I need to do is update `.bash_profile` with:

```
PATH="$PATH:~/jasp/"

export JASP=~/jasp/
```

Additionally, to make the assembler work successfully, you will need to ensure that the first line of the `jasm.pl` program uses the correct location for Perl on your system. You may also want to set up a logical file to point to the `jasm.pl` like this:

```
cd $JASP
ln -s jasm.pl jasm
```

A.2 The JASP Processor - A Reference

This section covers:

- A description of the function of each register;
- The micro-instructions understood by the JASP control unit;
- A description of the memory map, including a description of the memory-mapped peripherals;
- The interrupt mechanism;
- The file formats used by JASP for both instruction sets and machine code.

A.2.1 Registers

The processor registers are listed in table A.1.

Register	Width (bits)	Description
PC	16	Program counter - *is used to keep track of the memory address storing the next instruction to be executed*
INC	16	Incrementer - *is used to add one to the value held in the PC, something that needs to occur very often in most programs. Using the incrementer (effectively as a specialist register) is faster than using the ALU for this particular task, and importantly does not affect the PSR flags*
A	16	General Register A - *is the first of two general purpose registers, programmers can use the general purpose registers to store program bit patterns*
B	16	General Register B - *is the second of the two general purpose registers*
MAR	16	Memory Address Register - *is used as a specialist register to store the address of the memory location that we need to read from or write to*
MDR	16	Memory Data Register - *is used as a specialist register to store the data that we have just read from memory or need to write to memory*
IR	16	Instruction Register - *is the specialist register where we store the instruction once it has been fetched from memory*
ALUx	16	Arithmetic Logic Unit X Register *is the first of two specialist registers where we store bit patterns to be used in ALU operations*
ALUy	16	Arithmetic Logic Unit Y Register *is the second of two specialist registers where we store bit patterns to be used in ALU operations*
ALUr	16	Arithmetic Logic Unit Result Register *is the specialist register where the result from an ALU operation is stored*
SP	16	Stack Pointer - *is the specialist register used to store the address of the top of the stack held in memory*
PSR	16	Processor Status Register - *is where we store information about the state of the processor, including the state of the last ALU operation*

Table A.1 ***Processor registers***

A.2.2 Micro-Instructions

The JASP processor has a micro-programmed control unit, where each machine code instruction is defined as a sequence of micro-instructions known as a micro-program. These micro-programs are used by the control unit to execute individual instructions.

These micro-programs can be grouped together in an instruction set file, sometimes referred to as a microcode file. Two instruction set are distributed with the JASP toolkit, and it is a simple process to define and use new instructions.

All the micro-instructions that are recognized by the processor can be separated into one of four distinct micro-instruction groups. These are:

- Data movement micro-instructions;
- ALU micro-instructions;
- Test micro-instructions;
- Processor control micro-instructions.

All the micro-instructions within each group are described below.

Data Movement Micro-Instructions

There are over forty data movements, as listed in tables A.2 and A.3.

RTL	Notes
A←[MDR]	
A←[ALUr]	
A←[IR(operand)]	8-bit transfer, hi-byte of result = 0x00
B←[MDR]	
B←[ALUr]	
B←[IR(operand)]	8-bit transfer, hi-byte of result = 0x00
B←[PC]	
PC←[IR(operand)]	8-bit transfer, hi-byte of result = 0x00
PC←[ALUr]	
PC←[B]	
PC←[MDR]	
PC←[INC]	
INC←[PC]	
ALUx←[MDR]	
ALUx←[A]	
ALUx←[B]	
ALUx←[IR(operand)]	8-bit transfer, value is sign-extended

Table A.2 ***Data movement micro-instructions***

RTL	Notes
ALUx←[PC]	
ALUx←[ALUr]	
ALUx←[SP]	
ALUy←[A]	
ALUy←[B]	
ALUy←[MDR]	
ALUy←[IR(operand)]	8-bit transfer, value is sign-extended
MAR←[PC]	
MAR←[IR(operand)]	8-bit transfer, hi-byte of result = 0x00
MAR←[ALUr]	
MAR←[A]	
MAR←[B]	
MAR←[SP]	
MAR←[MDR]	
MDR←[A]	
MDR←[B]	
MDR←[ALUr]	
MDR←[ALUx]	
MDR←[PSR]	Moves flag contents
MDR←[M[MAR]]	Performs a memory read operation
M[MAR]←[MDR]	Performs a memory write operation
CU←[IR(opcode)]	8-bit transfer
IR←[MDR]	
SP←[ALUx]	
SP←[ALUr]	
SP←[MDR]	
SP←[IR(operand)]	8-bit transfer, hi-byte of result = 0x00
PSR←[MDR]	Restore 16-bit PSR contents
ALUy←[JUMPERS(IntBase)]	Transfer the interrupt base address to ALUy
ALUx←[PSR(IntVec)]	Transfer the interrupt vector to ALUx
PSR(IntVec)←[IR(operand)]	Set the interrupt vector
PSR(IntVec)←[MDR]	Set the interrupt vector with low 3-bits of MDR

Table A.3 ***Data movement micro-instructions - continued***

ALU Micro-Instructions

The ALU micro-instructions are listed in table A.4.

Code	Operation	RTL	Notes
0000	ADD	ALUr=[ALUx]+[ALUy]	Perform a 2's complement ADD operation, adding the ALUx and ALUy bit patterns together and storing the result in the ALUr register
0001	ADC	ALUr=[ALUx]+[ALUy]+[PSR(c)]	Perform a 2's complement ADC operation, adding the ALUx and ALUy and C flag together and storing the result in the ALUr register
0010	SUB	ALUr=[ALUx]-[ALUy]	Perform a 2's complement SUB operation, subtracting the ALUy from the ALUx bit pattern and storing the result in the ALUr register
0011	SL	ALUr=[ALUx]$<<$1	Perform a logical shift left on the ALUx, storing the result in the ALUr
0100	SR	ALUr=[ALUx]$>>$1	Perform a logical shift right on the ALUx, storing the result in the ALUr
0101	AND	ALUr=[ALUx]&[ALUy]	Perform a logical AND operation on the ALUx and ALUy bit patterns and storing the result in the ALUr register
0110	OR	ALUr=[ALUx]$\|$[ALUy]	Perform a logical OR operation on the ALUx and ALUy bit patterns and storing the result in the ALUr register
0111	NOT	ALUr=˜[ALUx]	Perform a logical NOT operation on the ALUx and storing the result in the ALUr register
1000	NEG	ALUr=˜[ALUx]+1	Perform a 2's complement negative operation on the ALUx and storing the result in the ALUr register
1001	INC	ALUr=[ALUx]+1	Add 1 to the ALUx bit pattern, storing the result in the ALUr
1010	DEC	ALUr=[ALUx]-1	Subtract 1 from the ALUx bit pattern, storing the result in the ALUr
1011	SWAP	ALUr(7:0)=[ALUx(15:8)]; ALUr(15:8)=[ALUx(7:0)]	Swap the most significant byte and the least significant byte of the ALUx, storing the result in the ALUr
1100	MUL	ALUr=[ALUx]*[ALUy]	Multiply the ALUx value by the ALUy value, storing the result in the ALUr
1101	DIV	ALUr=[ALUx]/[ALUy]	Divide the ALUx value by the ALUy value, storing the result in the ALUr
1110	MOD	ALUr=[ALUx]%[ALUy]	Divide the ALUx value by the ALUy value, storing the remainder in the ALUr

Table A.4 ***ALU micro-instructions***

Whenever an ALU operation is executed, the PSR flags V, N, C and Z are updated. Table A.5 shows how the flags are updated by each ALU operation - a key to this table is given in table A.6.

Operation	**V**	**N**	**Z**	**C**
ADD	*	*	*	*
ADC	*	*	*	*
SUB	*	*	*	*
SL	0	*	*	*
SR	0	*	*	*
AND	0	*	*	0
OR	0	*	*	0
NOT	0	*	*	0
NEG	*	*	*	*
INC	*	*	*	*
DEC	*	*	*	*
SWAP	0	*	*	0
MUL	*	*	*	0
DIV	*	*	*	0
MOD	*	*	*	0

Table A.5 ***How ALU operations affect the PSR flags***

Flag	**Meaning**
V	* means that if 2's complement overflow occurs then V=1 else V-0 (division overflow in cases of DIV and MOD) 0 means that V=0
N	* means N=MSB(ALUr)
Z	* means if (ALUr==0) then Z=1 else Z=0
C	* means if (carry from MSB of ALUr) then C=1 else C=0 * except with SR this means if (carry from LSB of ALUr) then C=1 else C=0 0 means that C=0

Table A.6 ***The key to figure A.5***

Test Micro-Instructions

The four PSR flags may be tested. If a test evaluates to TRUE, any remaining micro-instructions in that microprogram are executed. Otherwise the micro-instructions following the test are ignored.

The valid test micro-instructions are listed in table A.7.

RTL	Notes
if(PSR(c)==1)	Carry flag set
if(PSR(c)==0)	Carry flag clear
if(PSR(n)==1)	Negative flag set
if(PSR(n)==0)	Negative flag clear
if(PSR(z)==1)	Zero flag set
if(PSR(z)==0)	Zero flag clear
if(PSR(v)==1)	Overflow flag set
if(PSR(v)==0)	Overflow flag clear

Table A.7 ***Test micro-instructions***

Processor Control Micro-Instructions

The valid processor control micro-instructions are listed in table A.9.

ALU Connectivity To The Data Bus

The individual registers of the ALU are connected to the data bus via the ALU data bus connection circuitry. A 2-bit code is given to this circuitry to connect or disconnect ALU registers from the bus. The codes are listed in table A.8.

Code	Notes
`00`	All ALU registers disconnected from the bus
`01`	ALUx is connected to the bus
`10`	ALUy is connected to the bus
`11`	ALUr is connected to the bus

Table A.8 ***ALU connectivity to the data bus***

A.2.3 Memory

JASP has 8Kb of memory, accessed as 4096 16-bit words (addresses `$0000` to `$0FFF`). Some implementations of JASP can have extra memory installed. For example, JASPer can have a maximum of 65536 words of memory (addresses `$0000` to `$FFFF`).

An address points to a 16-bit word and all memory accesses are words. Note that this may not be the case with some popular microprocessors which have byte-addressable memories.

RTL	**Notes**
PSR(I)=0	Set the interrupt flag to 0
PSR(I)=1	Set the interrupt flag to 1
PSR(E)=0	Set the interrupt enable flag to 0
PSR(E)=1	Set the interrupt enable flag to 1
HALT	Processor halt
NOP	No operation

Table A.9 ***Processor control micro-instructions***

Two registers are associated with memory accesses. The Memory Data Register (MDR) contains the data value which is about to be written to memory or a value which has been read from memory. The Memory Address Register (MAR) contains the memory address of a read or write operation.

Think of MAR as a pointer to a word of memory. The pointer may be moved by altering the value held in MAR. Values may be transferred from the MDR to memory (Write) or from Memory to the MDR (Read).

To write a value into memory you do the following:

```
RTL                Description

MDR<-00FF          Place data in MDR
MAR<-0010          Place address in MAR
M[MAR]<-[MDR]      Update memory
```

Note the sequence of operations performed when writing data into memory. The address and data values are loaded into the MAR and MDR respectively and a write cycle is performed.

A memory read is as follows:

```
RTL               Description

MAR<-0010         Place address in MAR
MDR<-[M[MAR]]     Read memory
                  MDR now contains [M[0010]]
```

The memory map is shown in figure A.2.

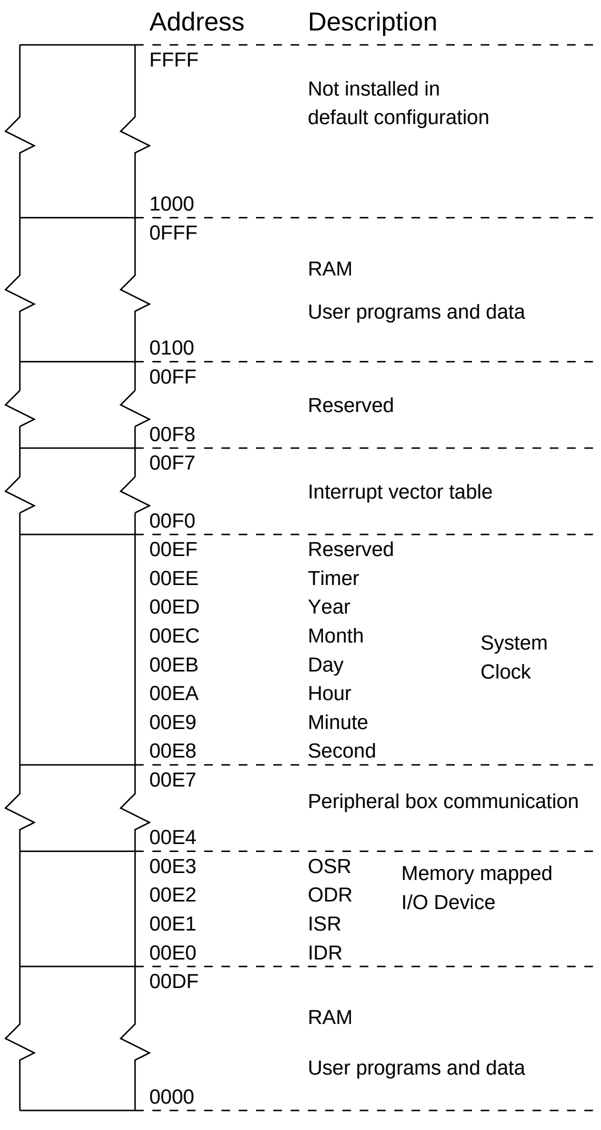


Figure A.2 ***The JASP memory map***

When accessing or writing to memory, all addresses are wrapped. So for example, if memory is installed up to `$0FFF`, then writing to the address `$1FFF` *will still cause memory to be updated*.

It can be seen that the small memory of the JASPER processor is used for a variety of purposes. The bulk of the memory is available for storing user data and instructions.

All peripherals have default locations within the memory map, but their locations are configurable.

Memory Mapped I/O

Handshaking needs to be used in order to perform I/O.

To write a character to the screen, first check that the OSR port is set to 1, if it's not go into a loop until it is. Only then write the character to the ODR.

To read a character from the keyboard, keep checking until the ISR is set to 1, only then should you read the character from the IDR.

Here is a piece of code that shows handshaking for both input and output:

```
* A demonstration of polled I/O
*
* This program reads 10 characters from the keyboard
* and then prints them all out once they've been entered
*
OSR      EQU   $E3       * Output Status Register (OSR)
ODR      EQU   $E2       * Output Data Register   (ODR)
ISR      EQU   $E1       * Input Status Register  (ISR)
IDR      EQU   $E0       * Input Data Register    (IDR)

         ORG   0
         MOVE  #$00,B    * count is storage for our
         MOVE  B,count   *   counter value

loop     MOVE  ISR,A     * Get ISR
         CMP   #$00,A    * is a char available?
         BEQ   loop      * no - wait some more
         MOVE  IDR,A     * read the char

         MOVE  count,B   * the address to write the
         ADD   #data,B   * value to is count+data
         MOVE  A,(B)     * write the char in there
         MOVE  count,B   * add 1 to count
         ADD   #$01,B    *
         CMP   #$0A,B    * and see if we have reached 10 and
         BEQ   gotchars  * move to next section if we have
         MOVE  B,count   * otherwise write count back
         JMP   loop      * and get another char
gotchars MOVE  #$00,B    * count is storage for our
         MOVE  B,count   *   counter value

write    MOVE  OSR,A     * get OSR
         CMP   #$00,A    * OSR 1 can print, OSR 0 can't print
         BEQ   write     * not yet, wait some more

         MOVE  count,B   * the address to read the
         ADD   #data,B   * value from is count+data
         MOVE  (B),A     * get the char in there
         MOVE  A,ODR     * print the char

         MOVE  count,B   * add 1 to count
         ADD   #$01,B    *
         CMP   #$0A,B    * and see if we have reached 10
         BEQ   done      * and move to end if we have
         MOVE  B,count   * otherwise write count back
         JMP   write     * and write another char
done     HALT            * done

count    DS.W $01        * the counter
data     DS.W $0A        * storage for our 10 characters
```

Current System Time

Additionally, JASP has a system clock device. The current date and time is accessible from `$00E8` to `$00ED`. No handshaking is required, simply access the particular memory location for the required date/time value. You cannot write to these memory locations, although no errors are raised if you try.

Peripheral Box Communication

A prototype peripherals board has been configured to work with the JASP processor - it was defined and built by Ian Chilton, under the supervision of Mark Burrell.

The peripherals board uses four memory locations, by default installed between $00E4 and $00E7, and the JASP processor can communicate with various input and output devices on the board. These include DIP switches, a buzzer, various LEDs and digit displays.

The peripherals board is purely a prototype for demonstration purposes only.

Reserved Addresses

It is recommended that you do not use the reserved memory addresses for storage - it could make your programs incompatible with future versions of JASP.

A.2.4 The Interrupt Mechanism

The interrupt mechanism makes use of an interrupt vector table stored in memory, and the I and E flags of the PSR.

The JASP processor can only deal with a single interrupt at any given time - any further interrupts generated while the first interrupt is being handled will be ignored. The actual details of the interrupt mechanism are definable within the instruction set. Within the default instruction set the interrupt mechanism is defined as:

```
PSR(I)=0                    interrupt flag = 0
MAR<-[SP]                     }     save PSR
MDR<-[PSR]                    }     on the stack
M[MAR]<-[MDR]                 }
ALUx<-[SP]                  }  decrement
ALUr=[ALUx]-1               }  SP
SP<-[ALUr]                  }
ALUx<-[PC]                    }
MDR<-[ALUx]                   }     write PC
MAR<-[SP]                     }     to the stack
M[MAR]<-[MDR]                 }
ALUx<-[SP]                  }  decrement
ALUr=[ALUx]-1               }  SP
SP<-[ALUr]                  }
PSR(E)=0                    interrupt enable flag = 0
ALUy<-[JUMPERS(IntBase)]      }
ALUx<-[PSR(IntVec)]           } build the vector address
ALUr=[ALUx]+[ALUy]            }
MAR<-[ALUr]                 } obtain the handler address
MDR<-[M[MAR]]               }
```

```
PC<-[MDR]                   load address of handler into PC
```

The position of the vector table is configurable, but it defaults to the locations $00F0 to $00F7.

A.2.5 JASP Files Reference

The JASP engine understands two file formats, these are the micro-instruction and machine code formats. In previous versions these formats proved to be somewhat stringent, and so have been made much more flexible.

Micro-Instruction File Format

Each micro-instruction file consists of a set of zero or more instruction definitions. These definitions begin with an `opcode` directive and an opcode, and then a set of micro-instructions to enter into the control unit micro-memory. Blank lines can exist anywhere within the file, and comments can be written after an asterisk. Each line in the micro-instruction file can be a maximum of 250 characters.

An instruction definition can also include two further directives, these are the `mnemonic` and `description` directives. The `mnemonic` directive describes the form of the mnemonic while the `description` directive gives a brief description of the opcodes function. Both the `mnemonic` and `description` directives expect their values to be within double quotes. Neither of these tags are mandatory, both being set to null strings if they are not included. If the line is not a directive and not a comment it is expected to hold a valid micro-instruction, followed by an optional comment.

A typical micro-instruction file might consist of a number of micro-programs each like the following instruction definition:

```
Opcode d0
* addr 00 to FF
Mnemonic "JSR addr"
Description "Jump to subroutine at a direct address"
ALUx<-[PC]             }
MDR<-[ALUx]            }     write PC
MAR<-[SP]              }     to the stack
M[MAR]<-[MDR]          }
ALUx<-[SP]                }  decrement
ALUr=[ALUx]-1             }  SP
```

```
SP<-[ALUr]                }
PC<-[IR(operand)]
```

Instruction Sets

JASP is provided with a default instruction set (instruct.mco). This instruction set contains all single word instructions where the hi-byte is the opcode. This set is limited to addressing memory in the range `$0000` to `$00FF`.

To use the full memory available, use the advanced instruction set in `advanced.mco`, however it is advisable to only use this instruction set in conjunction with the assembler rather than with hand coding.

Machine Code File Format

Each machine code file consists of a set of zero or more machine code segments. These segments begin with an `org` directive and address, and then a set of 16-bit values to enter into memory. Blank lines can exist anywhere within the machine code file, and comments can be written after an asterisk character. A simple program is shown below. Each line in the machine code file can be a maximum of 250 characters. If a line does not begin as a comment and does not have an `org` directive, then it is assumed to be a 16-bit value.

```
org 0400
45FF
3200
0001
F000   * this is the only comment in this program
```

A.3 JASPer

JASPer originally came into being as a clone of part of a software package for VAX/VMS known as *ASP*, or *Animated Simple Processor*, designed and written by William Henderson. The first Windows clone was known as *WinASP*, but when the VAX/VMS version was no longer supported then this package (which had grown into a package in its own right) became known as ASP. It has now changed its name (again!) to JASPer, to avoid any confusion with Microsoft ASP which is something totally different altogether.

So what is JASPer? It is a package that simulates the JASP processor. It can be used in one of two modes - either white-box mode where the internal registers of the processor are visible, or black-box mode where the user can see the output produced by their programs. What this figure does not demonstrate is that, when instructions are run (either individual micro-instructions or whole machine code programs) all data movements are animated within the package - graphically showing the inner workings of the processor.

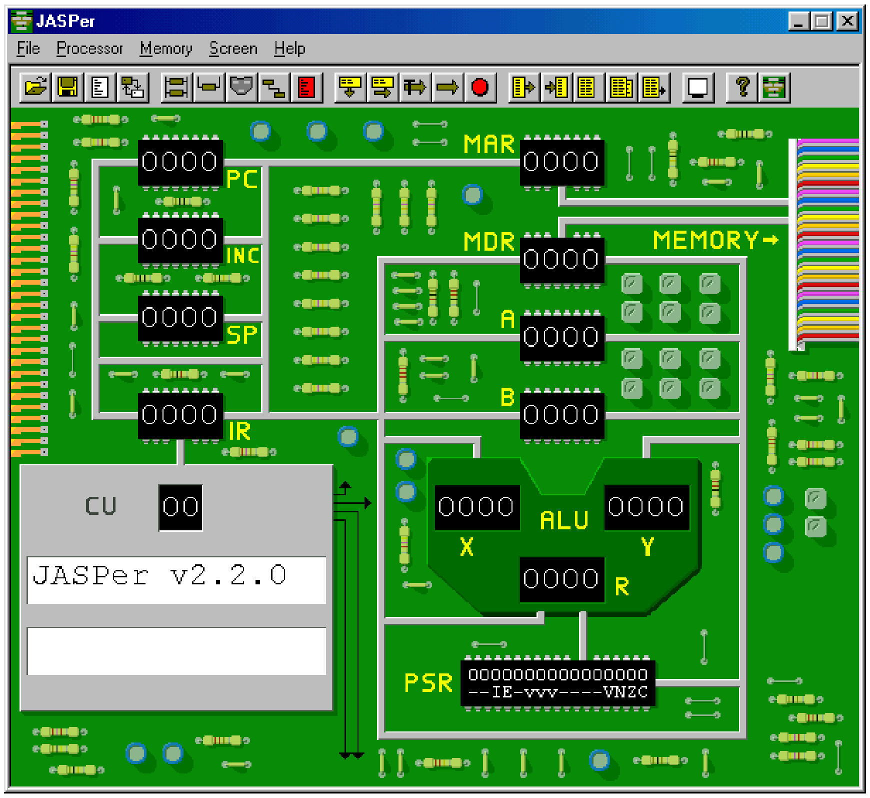
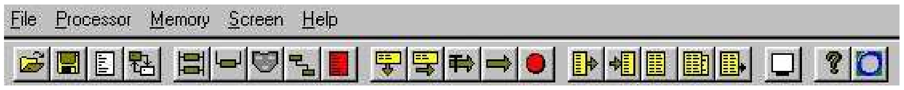


Figure A.3 ***JASPer - the main graphic display***

There are actually two different white-box views of the processor that can be selected. The first, as illustrated in figure A.3 is the main animated display. A second, simpler display, can be used instead - it is used for a number of screen-shots in the main text of the book - the simpler display is shown in figure A.4.

A.3.1 How To Use JASPer

The functionality of JASPer is accessed via the menu bar and the buttons below the menu bar:



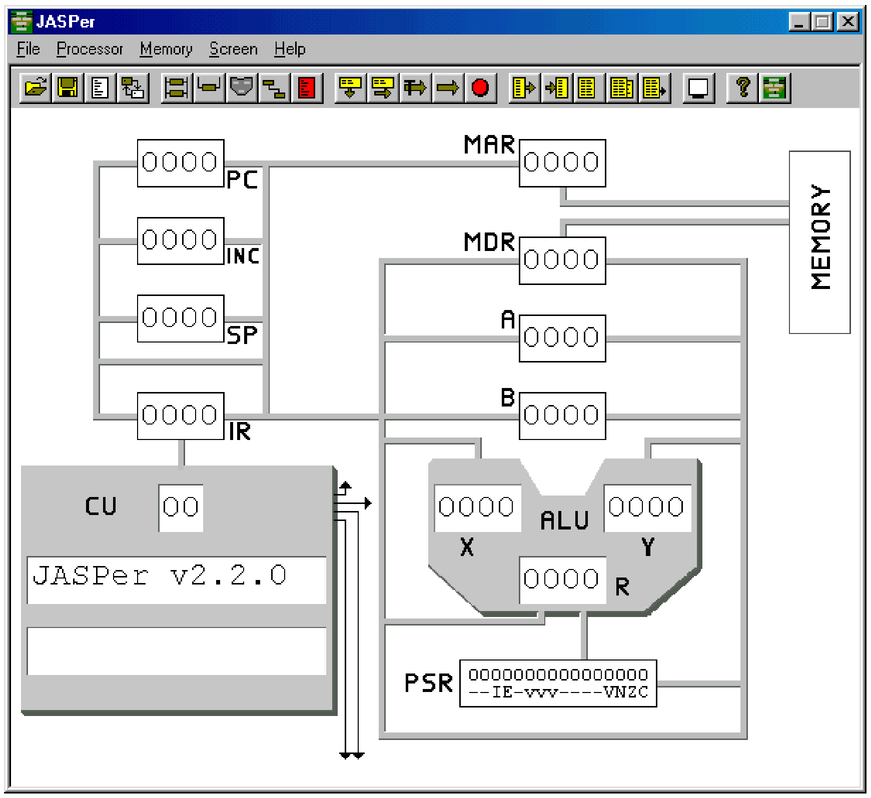


Figure A.4 *JASPer - the simple display*

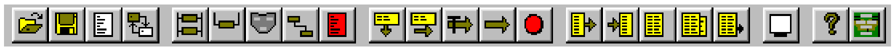
The Menu Bar

The menu bar has five entries. These are for controlling the processor, memory functions, etc. The menu bar looks like this:

As you can see, each entry has one letter underlined, pressing this letter together with the ALT key is the keyboard equivalent of clicking on the entry. For example, typing ALT-M will bring up the memory menu.

The Buttons

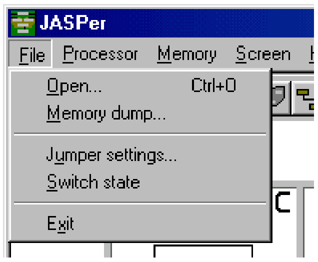
The buttons provide the same functionality as the menu bar, only in a graphical manner.



Usage

Both the menu bar and the buttons can be broken down into the same five key function areas, and these are now described in turn.

The File Menu

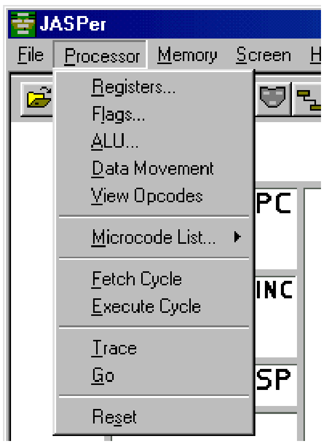


The file menu accessed from the keyboard provides the same functionality as the following set of buttons.

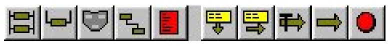
Button	Menu Option	Description
	Open	This option brings up a file open dialogue, allowing the user to open either a microcode file (MCO), or a macrocode file (JAS).
	Memory Dump	Save JAS file containing all the contents of memory.

Button	Menu Option	Description
	Jumper Settings	This allows the user to control the animation features, including animation speed.
	Switch State	This option switches between black-box mode and white-box mode. JASPer always starts in white-box mode, where the registers and data paths of the processor can be seen. In black-box mode the JASPer window switches to a view of JASPer's output - any I/O output will be seen here.

The Processor Menu



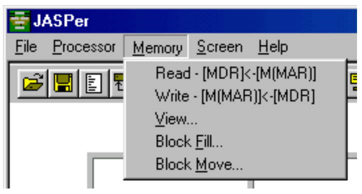
The processor menu is equivalent to the following set of buttons:



Button	Menu Option	Description
	Registers	Brings up a dialogue where the user can change register values.
	Flags	Brings up a dialogue where the user can change flag values.
	ALU	Brings up a dialogue where the user can perform ALU operations.
	Data Movement	Allows the user to perform all data movement operations. Click on the destination register first, followed by the source register in order to perform the data movement.

Button	Menu Option	Description
▤	View Opcodes	This option displays a summary of all the currently loaded opcodes loaded into the processor. There are three parts to the display, the opcode, the mnemonic and the brief description.
	Microcode List	This option doesn't have an equivalent button - it combines both the ALU and data movement operations. On using this option a further menu is displayed, allowing the user to select a particular micro-instruction to run - divided into ALU and data movement micro-instructions. The data movement micro-instructions are displayed by destination - one can select the destination and then an appropriate source register.
⇩	Fetch Cycle	This option runs a single fetch cycle.
⇨	Execute Cycle	This option runs a single execute cycle.
⇉	Trace	This option runs one fetch-execute cycle
⇨	Go	This runs the processor. The processor stops when either the Escape key (acting as a reset button) is pressed, the mouse is clicked on the JASPer window, or a halt instruction is executed. Note that, if animation is turned on, even if the Escape key or the mouse is clicked on the JASPer Window, the animation displays until either the current fetch or the current execute cycle completes.
●	Reset	This option resets the processor, as if the processor power switch has been cycled.

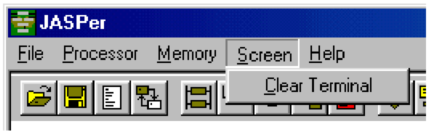
The Memory Menu



The memory menu accessed from the keyboard provides the same functionality as the following set of buttons.

Button	Menu Option	Description
	Read	Perform a memory read operation
	Write	Perform a memory write operation
	View	This option brings up a dialogue which allows the user to view the contents of memory. For each location it displays the address, value and mnemonic (of the opcode that is stored in the most significant byte of the memory address)
	Block Fill	This option brings up a dialogue that allows the user to fill user defined memory locations with a definable word value.
	Block Move	This option brings up a dialogue that allows the user to copy the contents of a given set of contiguous memory locations to another memory location. The original contiguous memory is unaffected.

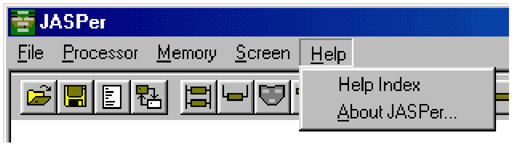
The Screen Menu



The memory menu accessed from the keyboard provides the same functionality as the following button.

Button	Menu Option	Description
	Clear Terminal	Terminal Clears the screen when JASPer is in black-box mode.

The Help Menu



The help menu accessed from the keyboard provides the same functionality as the following set of buttons.

Button	Menu Option	Description
?	Help Index	This option displays the Windows help file for JASPer.
🟩	About JASPer	This option brings up a dialogue that displays the current version of JASPer and build date, together with a copyright statement.

A.3.2 JASPer Parameters

Parameters for JASPER are:

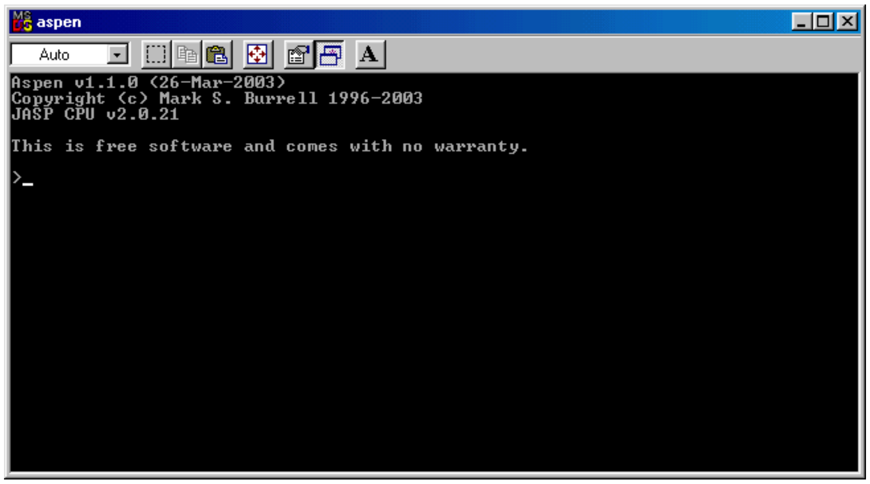
```
JASPER [/h*elp]

  or

JASPER [/l*oad={macrocodefile}][/m*co={microcodefile}] [/pc={addr}]
       [/n*odefault] [/bw] [/anim_off]
```

The meaning of each parameter is as follows:

```
/help        - displays the parameters for the program.
/load        - loads the given macrocode file
/mco         - loads the given microcode file
/pc          - sets the PC to the given value prior to running
/nodefault   - instructs the processor not to load the default
               instruction set
/bw          - use the simple display rather than the full
               graphic
/anim_off    - animation switched off by default
```

A.4 Aspen

Aspen is a command-line version of JASPer, obviously without any graphical display apart from a simple text output. Aspen actually uses the same processor engine as JASPer, and so the two programs share the (nearly) exact same functionality. Figure A.5 shows Aspen running in a Windows 95 DOS window. Versions of Aspen will run on any version of DOS (from V5.0 upwards), any version of Microsoft Windows and Linux (ELF binary).



```
Aspen v1.1.0 (26-Mar-2003)
Copyright (c) Mark S. Burrell 1996-2003
JASP CPU v2.0.21

This is free software and comes with no warranty.

>_
```

Figure A.5 ***Aspen***

When you use Aspen, you need to either make sure the default instruction set and the Aspen help file (`aspen.hco`) is in the same directory, or make use of the `JASP` environment variable as detailed previously. Using Aspen, and any text editor of your choice, it is possible to use even the lowliest 386 PC to develop programs for JASPer.

If you have the `JASP` environment set, then when you attempt to open either a microcode or a program file from within Aspen it will first attempt to find that file within the current directory, and if it can't find the file it will then look for it in the directory named in the `JASP` environment. Please note that the 16-bit version of Aspen only understands the 8.3 DOS naming convention.

Use the help facility within Aspen by typing `help` at the chevron prompt.

A.4.1 Aspen Parameters

Parameters for Aspen are:

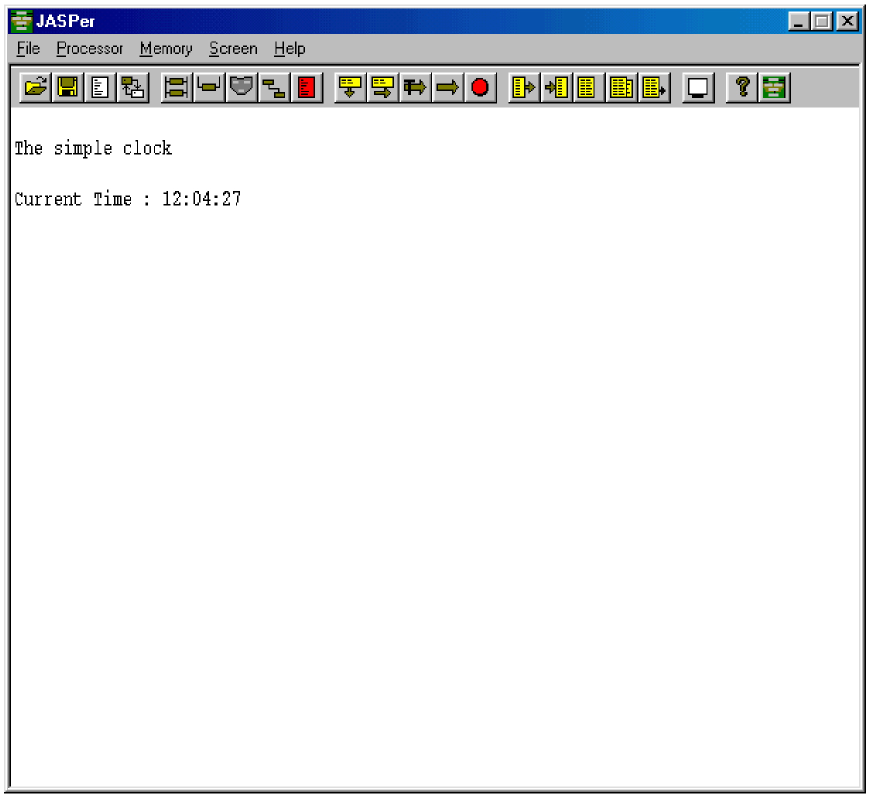
```
ASPEN [/h*elp]

  or

ASPEN [/l*oad={macrocodefile}][/m*co={microcodefile}] [/g*o] [/q*uit]
      [/pc={addr}] [/n*odefault]
```

The meaning of each parameter is as follows:

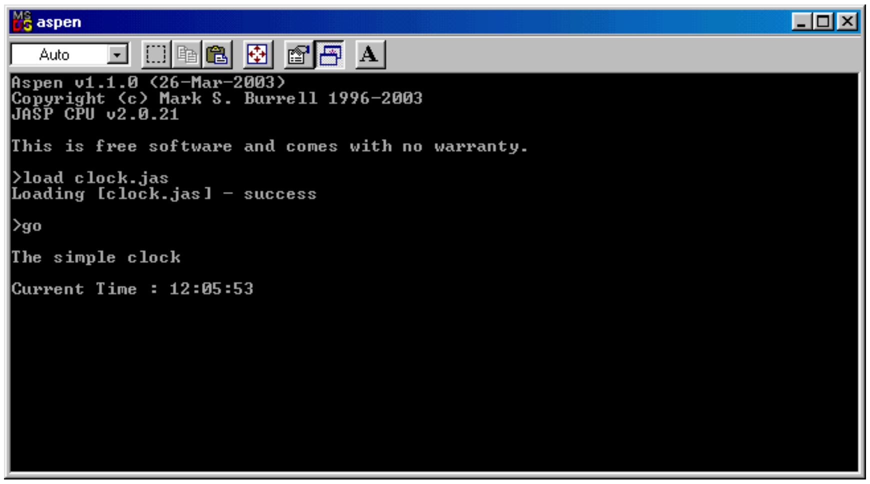
```
/help       - displays the parameters for the program.
/load       - loads the given macrocode file
/mco        - loads the given microcode file
/go         - instructs the processor to begin executing
/quit       - instructs the program to close once the given
              execution process has finished
/pc         - sets the PC to the given value prior to running
/nodefault  - instructs the processor not to load the default
              instruction set
```

A.4.2 Example Programs Running on the JASP

Here is the same program, *clock.jas* as distributed with the JASPer package, seen running in both JASPer and in Aspen.



```
The simple clock

Current Time : 12:04:27
```



```
Aspen v1.1.0 (26-Mar-2003)
Copyright (c) Mark S. Burrell 1996-2003
JASP CPU v2.0.21

This is free software and comes with no warranty.

>load clock.jas
Loading [clock.jas] - success

>go

The simple clock

Current Time : 12:05:53
```

A.5 The JASP Assembler

The JASP Cross-Assembler, to give it its full title, is a program written in Perl that can be used to ease the process of creating assembly language programs for the JASP processor.

The assembler should work on any system with a Perl installation, although it has only been test on DOS and Linux systems.

A.5.1 The Instruction Set Used

By default the assembler uses `instruct.mco` as the default instruction set. It then attempts to assemble your assembly language program using the information it finds within the instruction set file.

A.5.2 Directives

It understands a limited set of directives, listed here:

Directive	Example	Description
ORG	`org $0000`	Sets the program origin
DC.B	`DC.B 'hello',0`	Stores a byte in memory, can also be used to specify text strings
DC.W	`DC.W $1234`	Stores a word in memory
DS.W	`DS.W 1`	Defines storage in words
EQU	`IDR EQU $E0`	Defines a constant
MCO	`MCO "advanced.mco"`	Specifies the instruction set to use
LPC	`LPC $0100`	Sets the PC to a particular value
USE	`USE "advancedio.lib"`	Include a library file in the program

When a `USE` directive is encountered, the assembler attempts to load the file from the current directory, and if that fails then it attempts to load the file from the directory listed in the the JASP environment variable.

A.5.3 Usage

The options of the assembler are as follows:

```
jasm [-h]

  or

jasm [-m mco][-a asm][-l type][-o filename]
```

The meaning of each parameter is as follows:

```
-m mco      : loads a microcode file (can be multiple files
              separated by a ':')
-a asm      : loads a JASP assembler file
-l type     : output type can be default|debug|code|printout
-o filename : send assembler listing to filename listed
-h          : list this help information.
```

Additionally, the assembler understands a `force_32bit` directive that can be placed in instruction set files. This has the effect of forcing all machine code instructions to be written in 32 bits rather than the standard 16 bits.

A.5.4 Operand Sizes

The assembler, not understanding anything about assembly language, has to make assumptions about the bit length of operands. To do this it makes use of information within the instruction mnemonics.

For example, if the assembler sees something like `data`, `addr` or `dis` in a mnemonic then it assumes that this operand should be 8 bits wide. However, if the assembler sees something like `dataword`, `addrword` or `disword` in a mnemonic then it assumes that this operand should be 16 bits wide.

The assembler also makes sure that words begin and end on word boundaries, and will pad out any odd bytes, to form 16-bit words, with zeroes to ensure that all words are word aligned.

A.5.5 Error Messages

As the assembler has no real understanding of assembly language, but rather can only read an instruction set and output machine code following very strict rules, this means that often its error message leave a little to be desired. Please treat this as part of the learning process - after all, without the assembler you would have to encode every assembly language program by hand!

A.6 The JASP C−− Compiler

The JASP C−− cross compiler is written by David Harrison. The program, `jcc`, compiles programs written in a small educational language called C−− into assembly language files that can then be assembled by the JASP assembler. The name of the language is a pun on the language C++; David admits it isn't a great pun.

The syntax rules for high-level languages tend to be written in a form known as Extended Backus-Naur Form, or EBNF. Within EBNF, symbols are defined in terms of other symbols. For example, an `if` construct in C−− is described as:

```
if ::= 'if' '(' express ')' statement 'else' statement
       | 'if' '(' express ')' statement
```

This means that an `if` construct begins with the word `if` followed by a condition expression that is within brackets. If the condition is true then `statement` is executed (the definition of `statement` is elsewhere), and the statement after the `else` is executed if the condition is false. Alternatively, the `if` statement can omit the `else` section.

Lastly, any entries within square brackets, as shown below are optional.

```
variable dec ::= type ident [ '=' literal ]
```

The production rules for C−− are given here:

```
program ::= declarations compstat

declarations ::= { declaration ';' }

compstat ::= '{'  statement statements '}'

statements ::= { statement }

declaration ::= constant_dec | variable_dec

statement ::= compstat | assign | input | output | if | while

constant dec ::= 'const' type ident '=' literal

variable dec ::= type ident [ '=' literal ]

assign ::= variable '=' express ';'

input ::= 'cin' '>>' ident ';'

output ::= 'cout' '<<' express ';'

if ::= 'if' '(' express ')' statement 'else' statement
       | 'if' '(' express ')' statement

while ::= 'while' '(' express ')' statement

type ::= 'bool' | 'string' | 'int'

ident ::= letter ident_chars

literal ::= boolit | stringlit | intlit

letter ::= 'A' | .. | 'Z' | 'a' | .. | 'z'
```

```
ident_chars ::= { ident_char }

boolit ::= 'false' | 'true'

stringlit ::= '"' { printable } '"'

intlit ::= sign uint | uint

ident_char ::= letter | digit | '_'

printable ::= ' ' | .. | '~'     {ASCII codes 0x20 to 0x7F)

sign ::= '+' | '-'

uint ::= digit { digit }

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

express ::= basic relop express | basic

basic ::= term addop basic | term

term ::= factor mulop term | factor

factor ::= literal | ident | '(' express ')' | '!' factor

mulop ::= '*' | '/' | '%' | '&&'

addop ::= '+' | '-' | '||'

relop> ::= '<' | '<=' | '==' | '>=' | '>' | '!='
```

Using these rules we can compile programs such as the example below, provided by David:

```
// Computes the maximum, minimum, total and average of a list
// of integers read from input. The input is terminated by a
// 0 on the input.

int n       = 0     ;             // Number read in
int max     = 0     ;             // Maximum
int min     = 32767 ;             // Minimum
int total   = 0     ;             // Total
int count   = 0     ;             // Number of inputs
int average = 0     ;             // Average
bool done   = false ;             // Input done flag.
bool done2  = true ;              // Input done flag.
const string prompt               // Prompt for input
               = "Number  : " ;
const string mess1                // Output messages
               = "Maximum : " ;
const string mess2
               = "Minimum : " ;
const string mess3
               = "Total   : " ;
const string mess4
               = "Average : " ;
const string endl                 // End of line for output
               = "\n" ;
const string countzero
               = "count was zero" ;

{
  // Prompt for and read integer. If number is 0 we're done, otherwise
  // increment count, add number into running total, check if it's
  // the maximum or minimum so far and prompt for the next number.
  while (!done)
  { cout << prompt ;
    cin >>  n ;
    if (n == 0)
    { done = true ; }
    else
    { count = count + 1 ;
      total = total + n ;

      if (n > max)
      { max = n ; } ;
      if (n < min)
      { min = n ; } ;
    } ;
  } ;

  // Compute the average.
  if (count != 0)
  {
    average = total / count ;
  }
  else
  {
     cout << countzero;
```

```
      cout << endl;
   };

   // Output results
   cout << mess1 ;
   cout << max ;
   cout << endl ;
   cout << mess2 ;
   cout << min ;
   cout << endl ;
   cout << mess3 ;
   cout << total ;
   cout << endl ;
   cout << mess4 ;
   cout << average ;
   cout << endl ;
}
// End of program
```

The compiler doesn't take any parameters, instead re-direction is used to pass a C−− program to it, and the assembly language output can be re-directed into an assembly language program file.

For example, imagine that we have a C−− program called `myprog.c--` that we want to run in Aspen. Provided that the program compiles with no errors we would do the following:

```
jcc < myprog.c-- > myprog.asm
jasm -a myprog.asm -o myprog.jas
aspen /l=myprog.jas /m=advanced.mco
```

The first line uses the compiler to produce the assembly language program. In the second line we assemble this to the machine code program. The third line loads the program into Aspen, along with the advanced instruction set as used by all programs created with the compiler. Once loaded into Aspen (or JASPer) we can run the program.

Lastly, it is worth noting that the program `jcc` doesn't make use of the `JASP` environment variable, but then again it doesn't need to.

A.7 The JASP Software Libraries

Two libraries of subroutines are part of the JASP toolkit. The first library, `basicio.lib` offers a few subroutines that are useful for text input and output, and is only really intended to demonstrate the usefulness of libraries. It can only be used in relatively small programs if they are intended to fit into memory between `$0000` and `$00DF`.

The second library, `advancedio.lib`, is more useful and offers a number of extra subroutines not offered by `basicio.lib`.

To use a library in your assembly language program you need to use a `USE` directive as shown here:

```
         * include a library
         USE "basicio.lib"
```

A.7.1 The Basic I/O Library

The subroutines provided by `basicio.lib` are:

```
* putstring - prints packed strings, address has to be in register A.
* putchar   - prints a single character from lo-byte of register B.
* putword   - prints a word held as 4 hex chars (from A)
* putbyte   - prints a word held as 2 hex chars (from lo-byte of A)
* getchar   - get a character from the keyboard, char in A register.
* newline   - print a CR/LF pair
```

A.7.2 The Advanced I/O Library

The subroutines provided by `advancedio.lib` are superset of those offered by `basicio.lib`. This library is so large that it can only usefully be used in programs written for above `$0100` in memory using the advanced instruction set.

Here are the subroutines offered by this library:

```
* putdbyte   - prints lo-byte of register A as decimal value.
* putdword   - prints a word held as a decimal number
* putstring  - prints packed strings, address has to be in register A.
* putchar    - prints a single character from lo-byte of register B.
* putbyte    - prints a word held as 2 hex chars (lo-byte of A)
* putword    - prints a word held as 4 hex chars (from A)
* getchar    - get a character from the keyboard, char in A register.
* newline    - print a CR/LF pair
* getustring - read in an unpacked character string, address in A
*              and required size in B
* putustring - print an unpacked character string, address in A
* inkey      - read a character from the keyboard if available.
* getdword   - read up to 6 chars and interpret as a signed decimal value
* getbyte    - read 2 chars and interpret as a hexadecimal value
* getword    - read 4 chars and interpret as a hexadecimal value
* getdbyte   - read up to 4 chars and interpret as a signed decimal value
```