ATP

This document is a reference to ATP and not a general tutorial. It describes the system that ATP is simulating, and is a reference to the op codes. It refers to version 3.3 build 147

History of ATP

Assembler Training Program (ATP) was originated to support the hypothetical processor described in (Knott & Waites 2000). The original ATP was written in Visual Basic and was difficult to distribute and maintain. ATP V3 is a Java applet. As such it is straightforward to distribute: users simply open the web page; and easy to maintain: only the web page source needs to be changed for all users to have a new version.

The Simulated System

ATP is a simulation of a microprocessor and associated memory, and simple BIOS. It also incorporates an assembler program with a built in editor. The BIOS supports integer and character input and output; and string input. The processor can do a range of simple operations including basic integer arithmetic.

For experiments on interrupt driven i/o a single interrupt line and associated vector is provided.

Memory

The simulated processor is word addressed. This means that is the smallest element of memory that can be addressed is a 16_{10} bit word. The system has just over 1 k words of RAM, visible from the grid at bottom right of the ATP screen. Notice that the memory display column headers, row headers and cell contents are all in hex.

Interrupt vector

For most simple applications the interrupt vector can be ignored, and should not be changed unless you know what you are doing. This section can be skipped unless you plan to change the interrupt vectors

Registers

There are 14 registers.

Ten general purpose registers: `R0` to `R9`. `R0`, although a general register, is used by the software interrupts `SWI PUTCHAR`, `SWI PUTINT` and `SWI GETCHAR`.

`PC` is the program counter. This is increased by one for one-word instructions and two for two word instructions so that after execution of an instruction it points to the next instruction to be executed.

`VR` is he oVeRflow register. This holds the remainder after DIV (divide) and (possibly) the high order word on MUL (multiply) if the result does not fit into one word. If there is no remainder, or the result of a multiply does not need a high order word, then VR will contain zero after the operation.

`SP` is the stack pointer. If hardware interrupts are to be used this should be set to the top of stack. Conventionally $04FF_{16}$ is the top of stack but knowledgeable users may use other values (but take care!)

Flags

`C` Carry. Set if and only if previous operation sets the 17th bit.

`V` Overflow. Set if previous operation would have set any bit in notional high order word and clear otherwise. The register VR will hold the actual high order word result.

`Z` Zero. Set if previous operation would have resulted in all bits in 16 bit register being zero, otherwise unset

`S` Sign. Set if high order bit is set, i.e. if result of previous operation could be regarded as negative 16 bit signed number, otherwise unset

`I` Interrupt. If set then hardware interrupts can occur. If unset they cannot.

Memory map

Description	Syntax
0 - $04FF_{16}$	Available for user code and data. When processor starts PC is set to 0 so location 0 should contain an executable instruction (if only a jmp to somewhere else. If interrupts are used the top of this portion of memory is used as a stack, starting from 04FF and growing downwards.
xxx - $04FF_{16}$	Top of stack This is the recommend location for the top of the stack since it gives the most room to grow, however it can be located anywhere in user memory.. Stack grows downwards towards user memory. The register SP is the stack pointer.
0500_{16}	Keyboard interrupt vector. When a key is struck and if the contents of this location is not 0 the current value of PC is saved and the value in this location is loaded into PC. Hence there should be code to handle the keystroke at the location pointed to by the contents of 0500_{16} This handler code should be terminated by an iret instruction. If 0500_{16} contains 0 (the usual situation) then the built-in keyboard handler is used.
0510_{16}	The keyboard is mapped here. If this location is read the ASCII value of the most recently pressed key will be found. Reading this location sets location 0511_{16} to 0
0511_{16}	Key ready. Contains 0 if no key has been pressed since the location 0510_{16} was last read otherwise 1

Table 1. memory map

Syntax

In the following examples of syntax value stands for a number, regD for a destination register, regS for a source register, and reg for a register which can be source or destination or source and destination. Items in square brackets are optional.

Description	Syntax
Set contents of unnamed memory cell(s)	`DATA value[,value[,value]...]`
Set contents of named memory cell(s)	`someName DATA value[,value[,value]...]`
Set instruction counter	`ORG 27`
Label	`myLabel:`

Table 2. Directives - Executed at assemble-time and generate no code

Mode	Description	Syntax
Direct mode	Register to register operation	`OP regD, regS`
Immediate Mode	Register and literal number operation	`OP reg, @33`
Indexed mode	Load or store from/to an address determined from adding the number (33 in this example) to the content of the register	`OP regD, 33+regS`

Table 3. Addressing Modes

Entry	Syntax
Operating system call	`SWI NAME_OF_OF_CALL`

Table 4. Others

DATA

Can have just one value or comma delimited list. The `DATA` directive assembles values into memory at current assemble-time counter (see `ORG`). If optional label is used (to left of `DATA`) this is given the value of the location in memory used

Single characters in single quotes are converted to ASCII before assembly. Example:

```
mylabel DATA 65, ’B’, 66
```

ORG

As the assembler assembles the program in to memory it uses an internal counter to keep track of where the code for the next statement should be assembled. This starts from 0 by default. Users can control the value in the counter with the ORG directive. Example:

```
ORG 120
```

Sets the instruction counter to 120_{10}

Direct mode operations

Direct mode operations involve two registers. Example:

```
MOV R1, R2
```

Moves whatever is in R2 into R1

Immediate mode operations

Immediate mode opcodes involve one register and one literal number Example

```
MOV R1, @33
```

Moves 33_{10} into register 1

Index mode operations

Immediate mode opcodes involve one register and a memory cell whose address is calculated by adding a literal number to the value currently stored in a register. Only LDR (load register from memory) and STR (store register to memory) work with indexed mode.

```
LDR R4, 34+R1 ; load contents of memory location
; formed by adding R1 to 34
; into register R4
```

Loads whatever is in the memory address R1+34_{10} into register R4

Labels

A label is assigned the current value of the instruction counter at the point where the label is encountered. labels are used as the target of jumps. Example Inserts a label mylabel between the two mov statements. The values assigned to labels can be seen in the symbol table.

```
    MOV R1, R2
mylabel:
    MOV R3, R4
```

Inserts a label mylabel between the two mov statements. The values assigned to labels can be seen in the symbol table.

OP Codes

`MOV` Move contents of second operand into first. Direct and immediate mode.

`CMP` Compares second operand with first. Actually, it subtracts second from first and sets the flags but does not change either operand. Direct and immediate mode.

`MUL` Multiplies first operand by second and leaves result in first. If result is too big to fit in first then overflow flag is set and overflow is stored in VR. Direct and immediate mode.

`XOR` Xors first operand with second and leaves result in first. Direct and immediate mode.

`SUB` Subtract second operand from first and leaves result in first. Direct and immediate mode.

`LDR` Loads contents of effective address into first operand register.

`STR` Stores contents of first operand register into an effective address . Effective address can be a number or a number+a register. (Indexed mode)

`INC` Adds one to register.

`DEC` Subtracts one from register.

`JEQ` Jump if EQual (i.e. if Zero flag set)

`JNE` Jump if Not Equal (i.e. if Zero flag is clear)

`JGT` Jump if Greater Than

`JLE` Jump if Less than or Equal

`JLT` Jump if Less Than

`JVC` Jump if oVerflow Clear

`JVS`Jump if oVerflow Set

`IRET` Return from interrupt. Atomically (resets the interrupt flag, jumps to the address found at the top of the stack, and pops the stack)

`STI` Sets the interrupt flag, thus permitting interrupts to occur

`CLI` Clears (resets) the interrupt flag, thus preventing interrupts from occurring

`PUSH` Pushes contents of register onto the stack and decreses SP by one. (copies the contents of register into memory address pointed to by `SP`). The stack should be set up before executing `PUSH` or unexpected things will happen

`POP` Pops the stack into register and increases `SP` by one. (copies the contents memory address pointed to by `SP` into register). The stack should be set up before executing `POP` or unexpected things will happen

Operating system calls

There is assumed to be a very simple operating system to perform keyboard input and screen output. Programs need to be able to call into the operating system to do input and output. The opcode SWI <somefunction> calls into the operating system. In this processor all calls to the operating system pass in and get out data through R0. Thus, although R0 is a general register it is best to avoid using it so it is vacant for any operating system calls that need to be made.

```
mov r0, @5
swi putint ; put 5 out on screen (put unsigned int)
```

Outputs the value in R0 as an **unsigned** base 10 integer.

```
mov r0, @-5
swi putsint ; put -5 out on screen (put signed int)
```

Outputs the value in R0 as an **signed** base 10 integer.

```
mov r0, @65
swi putchar ; put A out on screen
```

Outputs the value in R0 as an ASCII character

```
swi getint
; R0 now contains the number the user entered
```

Gets an integer from the input window as a base 10 integer. The thread of execution stops and waits until the user presses the [input] button. If the number is preceeded by a minus sign it will be interpreted as a signed negative number. It the string can not be interpreted as a base 10 integer the message "PANIC: not a number" will be written in the status window and zero will be stored in R0. Execution will continue.

```
    jmp start ; jump over buffer to next executable line
    BUFFER data 5
    data " "
start:
    mov r0, @BUFFER ; move *address* of buffer into R0
    swi getstr
```

Gets a string from the user. Before the call R0 should be set to contain the address of a buffer, that is a series of addresses, large enough to contain the anticipated string. When the user types the string and hits return the buffer will be filled with the string formated as follows. The string format has the first (lowest) word as a number representing the string length (not including the first word), thus the string “ABC” would be represented in memory as

Address	Value	Description
55	3	Length word
56	65	ASCII "A"
57	66	ASCII "B"

Address	Value	Description
58	67	ASCII "C"

Table 5. String "ABC" in memory starting at 55

Numbers and characters

Unadorned numbers in programs are interpreted as base 10. Precede numbers with & to have them interpreted as hex (base 16). Although characters can be used in DATA statements (in single quotes), they can’t be used in immediate statements (this is a bug!)

Interrupt system

This is an advanced topic. By default interrupts are disabled and this section can be ignored.

There is a single simulated hardware interrupt line that is "raised" then a key is pushed. If interrupts are enabled, that is if the `I` flag is set, the current value of PC is pushed onto the stack, the `I` flag is cleared to prevent further interrupts, and a `jmp` is executed to the address stored in the keyboard interrupt vector (0500_{16}). The code at that address (not at 0500_{16} but at the address pointed to by 0500_{16}) is performed. This code should be terminated by an `iret` instruction, which pops the stored value of PC from the stack and executes a `jmp` to it and resets the `I` flag.

Code examples

Add two static numbers

```
MOV R1, @5
MOV R2, @3
ADD R2, R1
```

Moves 5 into R1; 3 into R2; and adds contents of R1 (5) into R2

Add two numbers from memory

```
ORG &10         ; start assembling at 10 hex (16 decimal)
N1 DATA 5       ; put 5 into memory address 10 hex and call it N1
N2 DATA 3       ; put 3 into memory address 11 hex and call it N2
ORG 0           ; go back to assembling at 0
LDR R1, N1      ; load R1 with contents of memory address N1
LDR R2, N2      ; load R2 with contents of memory address N2
ADD R2, R1      ; add R1 into R2
```

Stores 5 into 10_{16} and names it N1; Stores 3 into 11_{16} and names it N2. Loads contents of N1 into R1; contents of N2 into R2 then added R1 into R2 3

Loop and output

```
Loop
    MOV R1, @10 ; loop limit
    MOV R2, @0  ; loop start
TOP:            ; label
    MOV R0, R2
    SWI PUTINT  ; put whatever is in R0 out as a base 10 number
    INC R2      ; up the loop count
    CMP R2, R1  ; reached limit?
    JNE TOP     ; no - jump to TOP else drop out and end
```

Stores 5 into 10_{16} and names it N1; Stores 3 into 11_{16} and names it N2. Loads contents of N1 into R1; contents of N2 into R2 then added R1 into R2 3

Arrays

```
ORG &100
my_str DATA "hello world"
ORG &0
MOV R1, @3
LDR R0, my_str+R1
```

Is more or less equivalent to the high level code

```
char my_str[] = "hello world";
my_str[3];
```

Stores 5 into 10_{16} and names it N1; Stores 3 into 11_{16} and names it N2. Loads contents of N1 into R1; contents of N2 into R2 then added R1 into R2 3

Non-blocking input - polling

```
mylab:
    ldr r1, 1297  ; load r1 from 0x501
    cmp r1, @0    ; compare to zero
    jeq mylab     ; jump if zero
                  ; carry on if not zero
    ldr r0, 1296  ; load r0 from 0x500
    swi putchar   ; put r0 on screen as a character
    jmp mylab     ; go again
```

Non-blocking input - interuppt driven

```
org &500
  data &400

org 0
     sti
loop:
 mov r0, r1
 swi putint

 mov r0, @32
 swi   putchar

 inc r1
 cmp r1, @9
    jne loop

 mov r1, @0
    mov r0, @12
 swi putchar
    jmp loop

org &400
 mov r8, r0
 ldr r0, 510
```

```
   swi putchar
mov r0, r8
iret
```

Updates

This table of updates starts with build 146

Build number	Date	Changes
146	April 29, 2011	Added swi putsint to put out signed integer
		Added PUSH
		Added POP
147	May 2, 2011	Added IRET and tested interrupt driven I/O
		Fixed bug where memory location 0000 got set to 0000 as result of parsing ORG

Table 6. Updates