AC-16

PROGRAMMABLE
COMPUTER
REFERENCE

Automachef

TABLE OF CONTENTS

SYSTEM ARCHITECTURE

The AC-16 is a state-of-the-art Programmable Computer capable of managing up to four other devices via the output pins, named O0 through O3. All of the outputs are set by default in the off position when the computer is turned on or reset. It has four built-in signed 16-bit registers that can be used for any purpose, named V0 through V3. They are also called variables in this reference manual and they are capable of storing integer values between -32,768 and 32,767.

Some kitchen parts offer support for reading the number of actions they have performed, such as ingredient gates, assemblers and robotic arms. The number of actions they have performed since they were started can be retrieved by reading the variables I0 to I3. I0 will read the number of performed actions for the device connected to O0, I1 to O1, and so on. Parts that do not support reading this value (like for example conveyor belts) will always return 0.

The program memory of the computer can contain up to 32 lines of code written in the AC Assembly Language. The entire program is executed exactly 30 times per second, making it easy to program routines that require precise timing. And speaking of timing, by accessing the special read-only register TT you can access the current time of the day, in 24 hours format. So, if the current time is 3:45 p.m., the TT register will contain the 16-bit integer 1545.

There is a set of four special, read-only registers that will be automatically populated with information coming from the built-in order reader. Each one of these registers will contain a number greater than 0 if at least a new order of the specified type has arrived in the current 33 milliseconds tick and 0 otherwise. The number dictates how many new orders of that type have been received in the current 33 milliseconds tick. It can be referenced in AC Assembly Language instructions as R0 to R3.

The built-in order readers can also distinguish between orders

coming from different sources, such as a drive-thru window or customers ordering take-out food. For that purpose you can add the following letters as suffixes to the R0 to R3 variables:

> R for orders coming from the restaurant.
>
> T for orders coming from the take-out area.
>
> D for orders coming from the drive-thru window.

EXAMPLES

> Built-in Order Reader R2 set to detect Cheeseburgers.
>
> Variable R2R will contain the number of Cheeseburgers ordered from the restaurant.
>
> Variable R2T will contain the number of Cheeseburgers ordered from the take-out area.
>
> Variable R2D will contain the number of Cheeseburgers ordered from the drive-thru window.
>
> Variable R2 will contain the total number of Cheeseburgers ordered in the current 33 milliseconds tick.
>
> R2 will, therefore, contain the sum of R2R + R2T + R2D.

LANGUAGE REFERENCE

LABELS

Labels are named locations in the code that can be used to change the flow of execution via the jump (jmp/jne/jeq/jgt/jlt) instructions. They can only contain alphabetic characters and need to have between 1 and 10 characters in length followed by a colon.

EXAMPLES

```
loopagain:
```

```
belton:
```

```
endprogram:
```

ADD INSTRUCTION

The add instruction adds two values and stores the result in the variable specified in the third parameter. Remember, registers are 16-bit, so the result has to be in the range -32768 to 32767 to prevent errors.

SYNTAX

```
add <operand1> <operand2> <operand3>
<operand1> can be a variable or an integer value.
<operand2> can be a variable or an integer value.
<operand3> has to be a variable.
```

```
add V1 15 V2
```

```
add V0 V1 V0
```

CMP INSTRUCTION

The cmp instruction compares two values and sets the comparison register to either -1, 0 or 1. If the first value is smaller than the second one, it will be set to -1. If they are equal, it will be set to 0. If the first value is greater than the second one, it will be set to 1. The result can then be used to conditionally jump to a different part of the code using the jne/jeq/jlt/ instructions.

SYNTAX

```
cmp <operand1> <operand2>
<operand1> can be a variable or an integer value.
<operand2> can be a variable or an integer value.
```

EXAMPLES

```
cmp V1 30
```

```
cmp V1 V3
```

DEC INSTRUCTION

The dec instruction will decrement the specified variable by one, but it will never allow it to reach negative numbers, so it will automatically stop at 0. It is particularly useful to implement timers.

SYNTAX

dec <operand1>

<operand1> has to be a variable.

EXAMPLES

dec V0

dec V3

JMP/JNE/JEQ/JGT/JLT INSTRUCTIONS

All of these instructions jump to the specified label. jne stands for "jump if not equal", jeq stands for "jump if equal", jlt stands for "jump if less than" and jgt stands for "jump if greater than". They will jump to the specified label if the result of the last comparison to be done using the cmp instruction matches the result in the name of the instruction. For example, running a jne instruction after a comparison will make it jump to the specified label only if said comparison determined the parameters were not equal, a jgt instruction will

only jump if the first parameter in the comparison was greater than the second one and so on. The jmp instruction will always (unconditionally) jump to the specified label.

<u>SYNTAX</u>

```
jmp <operand1>

jne <operand1>

jeq <operand1>

jlt <operand1>

jgt <operand1>

<operand1> has to be a label.
```

<u>EXAMPLES</u>

```
jne endprogram
```

```
jlt loopagain
```

MOV INSTRUCTION

The mov instruction copies a value into the specified variable.

<u>SYNTAX</u>

```
mov <operand1> <operand2>

<operand1> can be a variable or an integer value.

<operand2> has to be a variable.
```

```
mov 30 V2
```

```
mov V1 V2
```

NOT INSTRUCTION

The not instruction simply toggles the value of a variable. If it was 0, it will become 1. Otherwise, it will become 0.

SYNTAX

```
not <operand1>
<operand1> has to be a variable.
```

EXAMPLES

```
not V1
```

```
not V3
```

OUT INSTRUCTION

The out instruction commands the device connected to the specified output to be turned on or off. If the second operand is 0, it will be turned off. Any other value will cause the output to be toggled to the on position.

```
out <operand1> <operand2>

<operand1> has to be an output.

<operand2> can be a variable or an integer value. 0 means off,
anything else means on.
```

Examples

```
out O2 1
```

```
out O2 V3
```

RET INSTRUCTION

The ret (means return) instruction finishes the execution
of code for this 33 milliseconds cycle. It has the same meaning
as jumping to a label placed right on the last line of code. It
has no operands.

Syntax

```
ret
```

Examples

```
ret
```

SUB INSTRUCTION

The sub instruction calculates the difference between two values and stores the result in the variable specified in the third parameter. Remember, registers are 16-bit, so the result has to be in the range -32768 to 32767 to prevent errors. Basically: it will do "operand1 - operand2" and store the result in the variable specified in operand3.

SYNTAX

sub <operand1> <operand2> <operand3>

<operand1> can be a variable or an integer value.

<operand2> can be a variable or an integer value.

<operand3> has to be a variable.

EXAMPLES

sub V1 15 V1

sub V2 V3 V0

EXAMPLE PROGRAM #1

This example program turns the device connected to O1 on for one second, off for one second and so on.

```
add 1 V0 V0

cmp 30 V0

jne endprogram

mov 0 V0

not V1

out O1 V1


endprogram:
```

EXAMPLE PROGRAM #2

This example program turns on the device connected to O2 after 5 orders have arrived.

```
add V0 R0 V0

cmp V0 5

jlt endprogram

out O2 1


endprogram:
```

EXAMPLE PROGRAM #3

This example program keeps the device connected to O0 on for 5

seconds for every new order that arrives. A good use case for this program is to make a dispenser produce one ingredient for every order that arrives. Note that this program will also prewarm the device for 4 seconds on startup, so ingredients are delivered almost immediately after the order arrives, saving precious time for your customers! A standard order reader can't do that, can it?

```
prewarm:
cmp V1 1
jeq alrdywarm
add 120 V0 V0
mov 1 V1


alrdywarm:
cmp R0 1
jne nonew
add 150 V0 V0


nonew:
cmp V0 0
jlt timerended
sub V0 1 V0
out O0 1
ret


timerended:
out O0 0
```

EXAMPLE PROGRAM #4

This complex example program will read from two different order reading modules (R0 and R1) and will handle outputs O0, O1 and O2. The goal of the program is to turn O0 and O1 on for three seconds whenever a new order arrives to either R0 or R1 but only turn on O2 when a new order arrives to R1. A good use case for this program is

to make R0 handle plain burgers and R1, cheeseburgers while O0 is connected to a raw patty dispenser, O1 to a burger bun dispenser and O2 to a cheese dispenser. It also prewarms the dispensers for two seconds on startup.

```
prewarm:
cmp V2 1
jeq checkorder
add V0 60 V0
add V1 60 V1
mov 1 V2


checkorder:
cmp R0 1
jeq addtime
cmp R1 1
jeq addtimes


main:
out O0 V0
out O1 V0
out O2 V1
dec V0
dec V1
ret


addtimes:
add V1 90 V1
addtime:
add V0 90 V0
jmp main
```