Part 2 – Emulator Development Guide

1 Introduction

Part 1 concerns the installation and operation of the emulator. This part is a guide to the operational details of the emulator to aid its further development.

Each of the sheets and the core cell functionality is described in this part. Some sheets contain buttons which activate macros and the macros are included in full with additional notes in the Appendix to this part. Many of the sheet cells have been allocated names to make the macros impacting their content easier to understand (and write in the first place).

Not every cell is described in the following. Some cells are copies or reformatted versions of other cells. These are easily seen by exploring the spreadsheet.

The spreadsheet cell formulae are largely compatible between Excel and Calc when importing the spreadsheet of the other type. The original development occurred on Excel. The formulae may contain subtle formatting differences and Apache documentation is available which explains the differences. The only incompatible cell formulae are contained in the Processor sheet at cells AK14 and 15. The returned values for the CELL function are different between Excel and Calc since the start row and column number references begin with one in Excel but zero in Calc. The difference is trivial and did not warrant a work-around during the development of the Calc version.

However, the macros are not compatible because Excel VBA and OpenOffice Basic contain core differences. Many functions are very similar but the macros are largely rewritten since it is not possible to import functional versions of the other type macros. The macros are the principal reason the spreadsheets can only function with the corresponding application program.

2 Processor Sheet

The cells which are updated by the macros need to be unprotected and hence are contained in a hidden area columns BA to BH. The register values are held here along with other modifiable data. All cells outside the hidden columns are protected and hence each protected register and memory cell is set by equating each protected cell to the corresponding hidden unprotected cell.

Control of the processor sheet appearance is summarised as follows:

- The values held by the registers – largely under the control of the macro “clock”. As each cycle is processed (with the button “clock”) the macro checks the cycle number and processes the register value accordingly.

- The content of memory – the visible area is formatted from two invisible tables (by using the same colour for font and background). The tables format the memory data into decimal and disassembled views from hidden content held in the column beginning “Memory_start”. The hex view is a conversion of the decimal view in the memory cell formulae. The hidden content may be set to zero (“Clear_Memory” macro) or copied from the assembled values from the “Assembler” sheet (“Load_Memory” macro).

- The control of the Gates – the text indicating the status of a Gate is derived from the instruction and cycle identified in the “Gates” sheet.

- The data flow animation – derived from some conditional cell content with conditional formatting according to the status of the Gates.

- The memory address decode shading – derived from the row and column hex numbers given by the hex address explicitly.

- The conditional branch animation – derived from the result of condition checking on the Processor sheet.

The instruction control properties as defined on the “Instruction Set” sheet for the current instruction are copied into the hidden area (column BF). The macro “clock” processing is controlled using the copied properties.

The 256 bytes of Memory data is copied from the Assembler sheet by the macro “Load_Memory”. The location on Processor is from BA25 (named “Memory_start”) to BA280. From here it is copied into the invisible table at AD25:AS40. The table is invisible because the font and background colours are the same. This technique applies to all results that are not part of the display.

The code is disassembled into column AU until the line number in AW corresponds to the “LOC_count”. This number is set to the “LOC_sum” on the Assembler sheet when the macro Load_Memory runs and determines that only the written program is disassembled. Column AV determines operand presence by a look-up to the “Instruction Set” sheet.

The disassembled code is rearranged in the invisible table AD42:AS57.

Clear Memory runs the macro “Clear_Memory” which copies the 256 zero-filled cells in column AC starting at AC25 into the corresponding cells starting at BA25 (Memory_start) and sets LOC_count to zero. The zeroes in cells AC25 to AC280 are invisible.

The displayed Memory is determined by the value “DHDSwitch” (BB16). The value in the cell is set by macros “Select_Hex”, “Select_Dec” and “Select_Dis” as 1, 0 or 2 respectively. Through each Memory cell formula the appropriate value is copied from the selected invisible table and converted to hex if DHDSwitch is 1.

Cell U21 displays the byte on the data line when the data bus is enabled (V15 = “Y”). This is copied from K44 which is a lookup into the row selected from the displayed memory and copied to K43:Z43. The lookups are through G43 and G44 in turn derived from the content of the Address Bus Register.

J25:J40, K41:Z41, K42:Z42 are used for conditional formatting around memory. If there is correspondence between any row and column address with the corresponding hex value of ADDRESS, then the result is one, otherwise zero. This includes the grey bars and black highlighting of the selected memory byte and the low hex digit of ADDRESS.

“mem_start_col” and “mem_start_row” (cells AK14 and 15) are used by the Clock macro to determine the column and row of Memory_start. The two cells are the only cells containing different formulae between the Excel and Calc types.

Cell J6 indicates the branch instruction type when Gate H is selected in the Gates sheet, otherwise the cell is blank. Content here signals a branch condition.

“BrDecision” (cell J5) decodes an entry in J6 into a value supporting branch processing. The instruction is identified and the branch condition tested. No branch instruction or no condition met results in the value zero. A branch instruction with the condition met results in the value one. Gate H is controlled by this value.

Cell G7 sets up a value according to the branch type placed in J6. The conditional formats of the paths from the Zero and Carry flags are set according to the values stored.

Cell J7 is blank until J6 contains content whereupon it indicates if the branch condition is met and Gate H is to be turned on (ENABLE) or otherwise (DISABLE). Conditional formatting along path to Gate H is controlled by J7.

3 Assembler

Text copied into the input columns of the sheet are processed in the hidden columns J to AD.

Columns AI to AT are also hidden. AJ, AK and AL contain the text copied by the macro “Assembler” from the “Source” sheet and so are unprotected and hidden. The displayed cells are a copy of the hidden version.

Labels are separated into “Constants” and “Variables” by the presence of “EQ” in the Instruction column.

An Instruction entry is marked as an “opcode?” if the content is not blank, “EQ” or “&”.

For each marked opcode? an “operand?” is marked if the column Op contains an entry. Memory markers “&” always include an operand (although there is no opcode). Otherwise there is no operand? marked.

The number of bytes “no. bytes” for each line entry is the sum of opcode? plus operand?.

“count” contains a running sum of the number of bytes in the program including memory bytes. “before” shows the value of “count” for the previous line i.e. the start position of the current line.

“value” is determined as the value in Op if the label is a “constant” or is “before” if the label is a “variable”. (“count” contains the number of bytes after adding the line entries). “value” is where an entry in Label is assigned its value.

“opcode” searches the “Instruction Set” sheet for the entry in Ins and returns any found match Machine Code (in decimal). No match returns zero.

“operand” assigns the value to the operand byte. A blank entry assigns zero. “EQ” in Ins assigns zero (but no byte is generated). An entry preceded with “#” is treated as a string and the right two characters are selected for a hex to decimal conversion. Any format problem returns an error condition (#NUM!). Any other entry is treated as a label and column AJ (labels) is searched for a match. A match returns the line entry under “value” (i.e. the value assigned). No match returns zero. Note that duplicate labels return the sum of the values and so are incorrect for all instances.

All bytes in the program are now contained in “opcode” and “operand”. Columns H and I (“Code”) display the entries with the address of the opcode in G (“Address”) derived from column X (“Byte#”).

Each entry in “opcode” and “operand” is assigned a byte number under “Byte#” for each column. This is achieved by a running sum over the entries in “opcode?” and “operand?”.

Column AA searches for the byte number entry in column Z in the opcode Byte# column. A match returns the opcode. Similarly, column AB searches the operand Byte# column and returns any match. Column AC combines the two columns to provide the final assembled code.

Column AO determines what entry in the Op code is mandatory. Column AN identifies the Instruction Codes with operands. For Instruction Codes with operands in Ins, a missing entry in Op returns the value one, otherwise blank. For Instruction Codes without operands an entry in Op returns the value one, otherwise blank.

Columns AP and AQ mark the presence of Instructions and where appropriate operands over the memory space. The number of bytes for each is summed at the bottom and their sum in the cell to the right “LOC_sum”. “LOC_sum” is used to determine which bytes are to be disassembled in the corresponding view on the Processor sheet.

Column AR and AS detect and number labels for inclusion in the Label Table. The inclusion of the index AT within AS returns the content of the Table in AU and AV. The Table is formatted using a conditional format.

Column J “Error check” deals with errors. Column AO is checked and a non-blank entry returns the value one. Column AK (Ins) is checked and blank, “EQ”, “&” and “HLT” removed from the test. For other entries column U is checked and should contain a recognised instruction (i.e. entry is non-zero). A zero entry can only occur if the instruction entry is not recognised and the returned entry in “Error check” is one.

Column F contains invisible text only made visible by conditional formatting when an entry in “Error check” is not blank.

Column AI contains an error check which returns “TRUE” if a spreadsheet error occurs during the conversion of a hex to decimal number. Otherwise the entry is “FALSE”.

Column AH contains invisible text only made visible by conditional formatting if the entry in column AI is “TRUE”.

4 ALU Acc

An entry into Processor sheet cell J9 (ALU instruction) is copied into “ALU Acc” sheet cell C3 which highlights the ALU section of ALU Acc through the conditional formatting.

In cell C26 the Instruction Code (mnemonic) at Processor sheet cell BF6 is compared to the mnemonics listed in ALU Acc sheet cells E30:E33 and a match copied. Conditional formatting highlights the Accumulator part of the sheet when C26 is not blank.

Additional conditional formatting occurs for blocks of cells around AND, OR, XOR, ADD and SUB portions where the names match the entry into C3. This highlights the current instruction calculations because the sheet always calculates all entries in the ALU for each ALU instruction.

Similarly, additional conditional formatting occurs in the Accumulator portion around the SHL, SHR, ROL and ROR instructions.

The hidden cells “Shift_Acc” and “Carry_in” in columns V and W are set to the values of the Accumulator and Carry flag during any fetch instruction by the macro Clock so that the ALU Acc sheet input values are stable to the end of the execution cycle (the Processor values are likely to change).

Other cell activity can be viewed directly in the spreadsheet.

5 Instruction Set

This sheet is at the heart of the emulator function. The instructions are defined here along with their characteristics as defined by the column headings (blank entries are effectively “N”). The column entries F to R determine how the instruction is processed in the emulator. Column D determines the number of processing cycles for each instruction. The other columns are largely informative although there is some processing according to the mnemonic in branching and on the “ALU Acc” sheet.

The summary of instruction entries over the entire code space (0 to 255) to the right is derived automatically by the hidden columns T, U and V.

On loading instructions into the Instruction Register the properties are copied using look-up formulae into the hidden part of the Processor sheet and are core to the operation of the Clock macro.

6 Micro code

This sheet is documentation and provides no calculation role in the operation of the emulator.

The codes contained in the “Microcode ROM Output” columns show the data decoded from the Instruction Register as described in Book 1 Part 2 for Fetch (0) and Execute (1 to 6) cycles. The low hex digit contained in the Fetch cycle ROM output contains the number of cycles in the instruction. This digit is loaded into a counter as described in Book 1 Part 2. The counter decrements after each cycle until it reaches zero, whereupon the instruction Execute cycle is completed.

The eight-bit Machine Code decodes the high eight bits of an 11-bit address on the ROM. The low three bits are decoded from the counter. The address of the resulting eight-bit blocks is given by multiplying the Machine Code by eight (i.e. left shift three bits) and is shown in Column L “Opcode x 8”. The first byte is the Fetch cycle output. The second byte is determined by the first byte address plus the count less one, since the counter is decremented at the end of each cycle.

The “Content of ROM” shows the final data programmed into the ROM that delivers the required control lines. See Book 1 Part 2. The content is derived from the hidden columns T to AR and AU.

7 Gates

The Gates sheet acts as the look-up map for Gate operation for all cycles within all instructions. The sheet provides for up to six cycles for each instruction, including the Fetch cycle.

The left set of columns labelled I to H contain the defining map which determines when a Gate is open. That is, if the corresponding Gate is to be open in a particular cycle as indicated by the cycle number 1 to 6, then the map entry is “1”. Blank entry indicates a closed Gate. Cycle number 1 is Fetch.

Column BI shows a marker at the instruction present in the Instruction Register at any time. Cell BI1 contains the value of the cycle currently selected (Fetch = 1).

The right set of columns I to H map the cycle number with the instruction and copy the mapping across where they correspond. At the base of the columns (the row above the repeated Gate headings) the map columns are added. The sums of the added six cycle columns for each Gate appear in the bordered boxes beneath the repeated Gate headings. An entry in a box indicates the corresponding Gate is open on the current cycle. The status of the Gates on the Processor sheet relate directly to the bordered boxes.

8 Concluding Remark

The spreadsheet and macro developments evolved over time and not on the basis of a specification or a complete set of requirements. Therefore, a simpler and more elegant implementation is undoubtedly possible and left as an exercise.

APPENDIX: MACROS

The macros for each emulator type are shown in the following with notes describing their function. The macros are as follows

Macro Name	Button	Notes

Clock	Clock	Each click advances the emulator by a Machine Cycle and processes the fetch/execute cycles. The contents of registers are updated.
		
		The macro contains a number of nested "If" statements. To make reading the macro simpler the nested level of the statements is colour-coded here as follows.
		If (level 1)
		  If (level 2)
		  If (level 3)
		  End If
		  End If
		End If
		
		Coloured ElseIf statements occur between the start and end at each level as appropriate. Statements which are easy to read are not colour-coded.

Reset	Reset	Sets register values to start values, usually zero. Forces the Program Counter to start the program from the beginning.
Assembler	Assemble	Copies the Source code into the assembler sheet for assembly.
Load_Memory	Load Memory	Copies the assembled output code into the Processor memory.
Clear_Memory	Clear Memory	Clears the Processor memory to zeroes.
Clear_Source	Clear Code	Fills the editable cells of the Source sheet with blanks.
Select_Hex	Hex	Sets the Processor memory content view to hexadecimal.

Macro Name	Button	Notes
Select_Dec	Decimal	Sets the Processor memory content view to decimal.
Select_Dis	Disassemble	Sets the Processor memory content view to instruction mnemonics and hex data.
CreateFile	Create Firmware	Creates a 256 byte binary file containing the byte output of the Assembler sheet for use by the simulator (Book 3).

1. EXCEL

Sub Clock() ' ' Increment the clock cycle count indicator ' Assume Execute cycle - if it is Fetch this will be over-written ' [clock_cycle] = [clock_cycle] + 1 [Fetch] = "N" [Execute] = "Y"	Clicking the “Clock” button runs this program. Each click is counted as clock_cycle. Most of the time the program is processing Execute cycles, so this is set here as the default. If it is actually a Fetch cycle, then this will be corrected in the following.

'
' Check the LDI_flag
' If set increment the Programme Counter and reset the flag
' This adjustment compensates for ease of viewing PC during LDI
'
If [LDI_flag] = "Y" Then
[Program_Counter] = [Program_Counter] + 1
[LDI_flag] = "N"
End If
	The emulator differs from the simulator (and a practical design) when the Program Counter actually increments. In the simulator this occurs at the end of the cycle. For clarity, the emulator animation shows the Program Counter – Address Bus Register – Memory flow and the PC is incremented at the start of the next cycle. LDI is the only instruction where the PC increment occurs in the last Execute cycle so the special flag LDI_flag is used to increment the PC before the next instruction is fetched. It is set when LDI is detected below.

'
' Increment the Machine Cycle count
' If the Execution is complete, reset the count to 1
'
[MC_count] = [MC_count] + 1
If [MC_count] > [MC_cycles] Then
[MC_count] = 1
End If
	Each increment of the clock increments the Machine Cycle count. If the MC count exceeds the number of cycles required by the instruction then execution is completed and the MC count is forced to one.

' ' Check for Fetch cycle MC Count is 1 ' If [MC_count] = 1 Then [Fetch] = "Y" [Execute] = "N" [Add_Bus_Reg] = [Program_Counter] Row_Address = [mem_start_row] + [Add_Bus_Reg] Col_Address = [mem_start_col] [Ins_Register] = Cells(Row_Address, Col_Address)	Check if the MC count is one. If so, then the cycle is a Fetch, so adjust the flags in the cells Fetch and Execute. Fetch cycle processing is as follows: Set the cell Add_Bus_Reg to the value in cell Program_ Counter. Calculate the variable Row_Address from the start of the Machine Code given by the start (mem_start_row) plus the content of Add_Bus_Reg. Set the Col_Address. Set the cell Ins_Register to the content of the cell calculated.
' Also set-up Accumulator value in Shift register and Carry [Shift_Acc] = [Accumulator] [Carry_in] = [Carry_flag]	Set the Accumulator and Carry flag into cells held by the ALU Acc sheet. This preserves the integrity of the sheet when the values change in the Processor. These are only significant for ALU Accumulator instructions. Following the Fetch processing the further macro Elseif checks are false and the macro ends.

```
'
' Other MC Count is Execute cycle
'
'
' Immediate Instructions
'
  ElseIf [Immediate] = "Y" Then
     [Program_Counter] = [Program_Counter] + 1
     If [MC_count] = 2 Then
          [Add_Bus_Reg] = [Program_Counter]
          Row_Address = [mem_start_row] + [Add_Bus_Reg]
          Col_Address = [mem_start_col]
       If [Load_ins] = "Y" Then
          [Accumulator] = Cells(Row_Address, Col_Address)
          [LDI_flag] = "Y"
       ElseIf [ALU] = "Y" Then
          [ALU_Mem] = Cells(Row_Address, Col_Address)
          [ALU_Acc] = [Accumulator]
          [ALU_Result] = [ALU_out]
          If [Add_flag] = "Y" Then
             [Carry_flag] = [Carry_out_add]
          ElseIf [Sub_flag] = "Y" Then
             [Carry_flag] = [Carry_out_sub]
          End If
       End If
```

If MC count is not one the cycle is an Execute cycle. The Immediate flag is checked. If the flag is set the following processing occurs after which the macro ends.

The PC is incremented for each clock entry of the immediate instruction. As described before, the emulator increments the PC at the start of processing, so occurs here for MC count 2 and 3.

For MC count equals 2 the operand cell address is calculated and processing occurs separately for the LDI and ALU instructions.

For LDI the cell content is copied to the Accumulator. The LDI flag is set so that the PC is incremented on the next clock because no further entry here occurs for LDI (MC_count is equal to MC_cycles at this point).

For ALU the cell content is copied to the ALU_Mem cell. The Accumulator is copied to the ALU_Acc cell. The ALU Acc sheet provides the result for the appropriate arithmetic/logic instruction in ALU_out which is copied into ALU_Result.

For arithmetic instructions the appropriate output carry is copied to the Carry flag.

ElseIf [MC_count] = 3 Then
  If [ALU] = "Y" Then
  [Accumulator] = [ALU_Result]
  End If
End If
	For an immediate instruction of three Machine Cycles the code following the test here is processed when MC count equals three. This can only be an ALU instruction in this design (but is tested anyway) and writes the result of the ALU into the Accumulator.

'
' Memory operation instructions
'
ElseIf [Load_Address] = "Y" Then
  If [MC_count] = 2 Then
  [Program_Counter] = [Program_Counter] + 1
  [Add_Bus_Reg] = [Program_Counter]
  Row_Address = [mem_start_row] + [Add_Bus_Reg]
  Col_Address = [mem_start_col]
  [Address_Register] = Cells(Row_Address, Col_Address)
	If MC count is not one and the instruction is not immediate, a check is made for an instruction operating memory. This includes load, store and ALU instructions.
	
	For MC count equals 2 the PC is incremented and the Address Register loaded with the operand value using the same address calculation seen in the above.

ElseIf [MC_count] = 3 Then
  [Program_Counter] = [Program_Counter] + 1
  If [Indirect] = "Y" Then
  [Add_Bus_Reg] = [Address_Register]
  ElseIf [Load_ins] = "Y" Then
  [Add_Bus_Reg] = [Address_Register]
  Row_Address = [mem_start_row] + [Add_Bus_Reg]
  Col_Address = [mem_start_col]
  [Accumulator] = Cells(Row_Address, Col_Address)
	For MC count equals 3 the PC is incremented.
	
	For indirect instructions the Add_Bus_Reg is set to the value in Address_Register. No further processing occurs for this cycle.
	
	For LDA the Accumulator is loaded with the value in memory using the same address calculation seen in the above.

ElseIf [Store] = "Y" Then [Add_Bus_Reg] = [Address_Register] Row_Address = [mem_start_row] + [Add_Bus_Reg] Col_Address = [mem_start_col] Cells(Row_Address, Col_Address) = [Accumulator]	For STA the address calculation is as seen before but the copy is from Accumulator to memory.

ElseIf [ALU] = "Y" Then
[Add_Bus_Reg] = [Address_Register]
Row_Address = [mem_start_row] + [Add_Bus_Reg]
Col_Address = [mem_start_col]
[ALU_Mem] = Cells(Row_Address, Col_Address)
[ALU_Acc] = [Accumulator]
[ALU_Result] = [ALU_out]
If [Add_flag] = "Y" Then
[Carry_flag] = [Carry_out_add]
ElseIf [Sub_flag] = "Y" Then
[Carry_flag] = [Carry_out_sub]
End If
End If
	For ALU instructions the address calculation is as seen before and the data is loaded into ALU_Mem. The Accumulator is copied into ALU_Acc and the result from the ALU Acc sheet ALU_out into ALU_Result.
	
	For arithmetic instructions the appropriate output carry is copied to the Carry flag.

ElseIf [MC_count] = 4 Then
  If [Indirect] = "Y" Then
  Row_Address = [mem_start_row] + [Add_Bus_Reg]
  Col_Address = [mem_start_col]
  [Address_Register] = Cells(Row_Address, Col_Address)
  ElseIf [ALU] = "Y" Then
  [Accumulator] = [ALU_Result]
  End If
	For MC count equals four the instruction is either indirect or an ALU operation.
	
	For indirect instructions the memory address to the data is calculated and loaded into the Address Register.
	
	For ALU instructions the result is copied to the accumulator.

ElseIf [MC_count] = 5 Then
  [Add_Bus_Reg] = [Address_Register]
  Row_Address = [mem_start_row] + [Add_Bus_Reg]
  Col_Address = [mem_start_col]
  If [Load_ins] = "Y" Then
  [Accumulator] = Cells(Row_Address, Col_Address)
  ElseIf [Store] = "Y" Then
  Cells(Row_Address, Col_Address) = [Accumulator]
  End If
End If
	For MC count equals five the instruction is an indirect memory operation.
	
	The address of the data is calculated. Data is loaded into the Accumulator, or stored in memory, according to a read or write instruction.

'
' Miscellaneous instructions
'
ElseIf [Shift] = "Y" Then
If [MC_count] = 2 Then
[Program_Counter] = [Program_Counter] + 1
[Accumulator] = [Shift_out]
[Carry_flag] = [Carry_out_shift]
End If
	The Accumulator instructions are processed here. The PC is incremented and the output from the ALU Acc sheet copied into the Accumulator and Carry flag.

ElseIf [Clear_Carry] = "Y" Then
If [MC_count] = 2 Then
[Program_Counter] = [Program_Counter] + 1
[Carry_flag] = 0
End If
	The Clear Carry instruction forces the Carry flag to zero.

ElseIf [Set_Carry] = "Y" Then
If [MC_count] = 2 Then
[Program_Counter] = [Program_Counter] + 1
[Carry_flag] = 1
End If
	The Set Carry instruction forces the Carry flag to one.

' ' Branch instructions ' ElseIf [Branch] = "Y" Then If [MC_count] = 2 Then [Program_Counter] = [Program_Counter] + 1 [Add_Bus_Reg] = [Program_Counter] Row_Address = [mem_start_row] + [Add_Bus_Reg] Col_Address = [mem_start_col] [Address_Register] = Cells(Row_Address, Col_Address)	Branch instruction execute cycle processing occurs here. For MC count equals two the Address Register is loaded with the instruction operand in the same way as seen before.

ElseIf [MC_count] = 3 Then
[Program_Counter] = [Program_Counter] + 1
If [BrDecision] = 1 Then
[Program_Counter] = [Address_Register]
End If
End If
End If
End Sub
	For MC count equals three the Branch condition result is checked and the PC updated by the Address Register if Gate H is opened.
	
	End of macro

Sub Reset()
  [clock_cycle] = 0
  [MC_count] = 0
  [Program_Counter] = 0
  [Add_Bus_Reg] = 0
  [Ins_Register] = 0
  [Accumulator] = 0
  [ALU_Result] = 0
  [Address_Register] = 0
  [Carry_flag] = 0
  [Carry_in] = 0
  [ALU_Acc] = 0
  [ALU_Mem] = 0
  [Fetch] = "N"
  [Execute] = "N"
  [LDI_flag] = "N"
End Sub
	Clicking the “Reset” button runs this program.
	
	The named cells are set to the value shown (0 or N).

Sub Assembler()
  Sheets("Source").Select
  Range("C2:E129").Select
  Selection.Copy
  Sheets("Assembler").Select
  Range("AJ2").Select
  Selection.PasteSpecial Paste:=xlPasteValues,
Operation:=xlNone, SkipBlanks _
  :=False, Transpose:=False
  Range("A18").Select
  Sheets("Source").Select
  Range("L2").Select
  Application.CutCopyMode = False
  Sheets("Assembler").Select
End Sub
	Clicking the “Assemble” button runs this program.
	
	This macro copies the content of the editable cells on the Source sheet into hidden cells on the Assembler sheet. Only the values are copied. The hidden cells are visible in the copy shown in the panel.
	
	The cells are deselected and the active sheet on macro end is the Assembler.
	
	The macro was constructed using macro recording.

Sub Load_Memory()
  Sheets("Assembler").Select
  Range("AF3:AF258").Select
  Selection.Copy
  Sheets("Processor").Select
  Range("Memory_start").Select
  Selection.PasteSpecial Paste:=xlPasteValues,
Operation:=xlNone, SkipBlanks _
  :=False, Transpose:=False
  Sheets("Assembler").Select
  Application.CutCopyMode = False
  Range("A18").Select
  Sheets("Processor").Select
  Range("B18").Select
  [LOC_count] = [LOC_sum]
End Sub
	Clicking the “Load Memory” button runs this program.
	
	This macro copies the content of the output cells on the Assembler sheet into hidden cells on the Processor sheet. Only the values are copied. The hidden cells are visible in the copy shown in the memory panel according to the selected format (hex, decimal or disassemble).
	
	The cells are deselected and the active sheet on macro end is the Processor.
	
	The macro was constructed using macro recording.
	
	LOC_count is added to limit the disassemble view to the instruction codes and not subsequent data.

Sub Clear_Memory()
  Range("AC25:AC280").Select
  Selection.Copy
  Range("Memory_start").Select
  Selection.PasteSpecial Paste:=xlPasteValues,
Operation:=xlNone, SkipBlanks _
  :=False, Transpose:=False
  Range("B18").Select
  Application.CutCopyMode = False
  [LOC_count] = 0
End Sub
	This macro copies the content of a range of cells set to zero on the Processor sheet into hidden memory cells on the Processor sheet. Only the values are copied. The hidden cells are visible in the copy shown in the memory panel according to the selected format (hex, decimal or disassemble).
	
	The cells are deselected and the active sheet on macro end is the Processor.
	
	The macro was constructed using macro recording.
	
	LOC_count is added to limit the disassemble view to the instruction codes and not subsequent data. It is set to zero.

Sub Clear_Source()
Range("F2:H129").Select
  Selection.Copy
  Range("C2").Select
  Selection.PasteSpecial Paste:=xlPasteValues,
Operation:=xlNone, SkipBlanks _
  :=False, Transpose:=False
  Range("A20").Select
  Application.CutCopyMode = False
End Sub
	This macro copies the content of a range of cells set to blank on the Source sheet into the editable panel on the Source sheet. Only the values are copied.
	
	The cells are deselected.
	
	The macro was constructed using macro recording.

Sub Select_Hex() [DHDSwitch] = 1 End Sub	Clicking the “Hex”, “Decimal” and “Disassemble” buttons run the respective program. The three macros set the value in cell DHDSwitch as showm. DHDSwitch is used in IF statements to control the memory view.
Sub Select_Dec() [DHDSwitch] = 0 End Sub	
Sub Select_Dis() [DHDSwitch] = 2 End Sub	

Sub CreateFile()
 	
Dim i As Long
Dim RowNum As Long
Dim ColNum As Long
 	
Open "Firmware.bin" For Binary As #1
 	
With ActiveSheet
  RowNum = .Range("Assemb_output").Row
  ColNum = .Range("Assemb_output").Column
  For i = 0 To 255
  Put #1, , CByte(Cells(RowNum + i, ColNum).Value)
  Next
End With
 	
Close #1
 	
MsgBox "Done!"
 	
End Sub
	Clicking the “Create Firmware” button runs this program.
	
	This macro creates a binary file for use by the simulator (Book 3).
	
	The macro opens or creates and opens the file “Firmware.bin” in the directory path defined by the “Default file location” set in the “General” tab of the spreadsheet options.
	
	The binary file is defined by the 256 bytes created from the 256 cells running below the cell “Assemb_output”.
	
	Once defined, the file is saved and closed.
	
	The message box provides positive feedback to the user that the task has been performed successfully.

2. CALC

Sub Clock() Doc = ThisComponent Processor = Doc.Sheets.getByName("Processor") ALU_Shift = Doc.Sheets.getByName("ALU Acc") Accumulator = Processor.getCellrangeByName("Accumulator") Add_Bus_Reg = Processor.getCellrangeByName("Add_Bus_Reg") Add_flag = Processor.getCellrangeByName("Add_flag") Address_Register = Processor.getCellrangeByName("Address_Register") ALU = Processor.getCellrangeByName("ALU") ALU_Acc = Processor.getCellrangeByName("ALU_Acc") ALU_Mem = Processor.getCellrangeByName("ALU_Mem") ALU_Result = Processor.getCellrangeByName("ALU_Result") Branch = Processor.getCellrangeByName("Branch") BrDecision = Processor.getCellrangeByName("BrDecision") Carry_flag = Processor.getCellrangeByName("Carry_flag") Clear_Carry = Processor.getCellrangeByName("Clear_Carry") clock_cycle = Processor.getCellrangeByName("clock_cycle") Execute = Processor.getCellrangeByName("Execute") Fetch = Processor.getCellrangeByName("Fetch") Immediate = Processor.getCellrangeByName("Immediate") Ins_Register = Processor.getCellrangeByName("Ins_Register") Instruction = Processor.getCellrangeByName("Instruction") LDI_flag = Processor.getCellrangeByName("LDI_flag") Indirect = Processor.getCellrangeByName("Indirect")	Clicking the “Clock” button runs this program. In OpenOffice Basic the objects used in the program have to be explicitly defined. The document, sheets and cells are explicitly defined here for the macro.

Load_Address = Processor.getCellrangeByName("Load_Address") Load_ins = Processor.getCellrangeByName("Load_ins") MC_count = Processor.getCellrangeByName("MC_count") MC_cycles = Processor.getCellrangeByName("MC_cycles") mem_start_col = Processor.getCellrangeByName("mem_start_col") mem_start_row = Processor.getCellrangeByName("mem_start_row") Program_Counter = Processor.getCellrangeByName("Program_Counter") Set_Carry = Processor.getCellrangeByName("Set_Carry") Shift = Processor.getCellrangeByName("Shift") Store = Processor.getCellrangeByName("Store") Sub_flag = Processor.getCellrangeByName("Sub_flag") zero_flag = Processor.getCellrangeByName("zero_flag") ALU_out = ALU_Shift.getCellrangeByName("ALU_out") Carry_in = ALU_Shift.getCellrangeByName("Carry_in") Carry_out_add = ALU_Shift.getCellrangeByName("Carry_out_add") Carry_out_shift = ALU_Shift.getCellrangeByName("Carry_out_shift") Carry_out_sub = ALU_Shift.getCellrangeByName("Carry_out_sub") Shift_Acc = ALU_Shift.getCellrangeByName("Shift_Acc") Shift_out = ALU_Shift.getCellrangeByName("Shift_out")	

Rem ' Rem ' Increment the clock cycle count indicator Rem ' Assume Execute cycle - if it is Fetch this will be over-written Rem ' clock_cycle.Value = clock_cycle.Value + 1 Fetch.setString("N") Execute.setString("Y")	Each click is counted as clock_cycle. Most of the time the program is processing Execute cycles, so this is set here as the default. If it is actually a Fetch cycle, then this will be corrected in the following.

Rem '
Rem ' Check the LDI_flag
Rem ' If set increment the Programme Counter and reset the flag
Rem ' This adjustment compensates for ease of viewing PC during LDI
Rem '
If LDI_flag.String = "Y" Then
Program_Counter.Value = Program_Counter.Value + 1
LDI_flag.setString("N")
End If
	The emulator differs from the simulator (and a practical design) when the Program Counter actually increments. In the simulator this occurs at the end of the cycle. For clarity, the emulator animation shows the Program Counter – Address Bus Register – Memory flow and the PC is incremented at the start of the next cycle. LDI is the only instruction where the PC increment occurs in the last Execute cycle so the special flag LDI_flag is used to increment the PC before the next instruction is fetched. It is set when LDI is detected below.

Rem '
Rem ' Increment the Machine Cycle count
Rem ' If the Execution is complete, reset the count to 1
Rem '
MC_count.Value = MC_count.Value + 1
If MC_count.Value > MC_cycles.Value Then
MC_count.Value = 1
End If
	Each increment of the clock increments the Machine Cycle count. If the MC count exceeds the number of cycles required by the instruction then execution is completed and the MC count is forced to one.

Rem ' Rem ' Check for Fetch cycle MC Count is 1 Rem ' If MC_count.Value = 1 Then Fetch.setString("Y") Execute.setString("N") Add_Bus_Reg.Value = Program_Counter.Value Row_Address = mem_start_row.Value + Add_Bus_Reg.Value Col_Address = mem_start_col.Value temp = Processor.getCellByPosition(Col_Address, Row_Address) Ins_Register.Value = temp.Value	Check if the MC count is one. If so, then the cycle is a Fetch, so adjust the flags in the cells Fetch and Execute. Fetch cycle processing is as follows: Set the cell Add_Bus_Reg to the value in cell Program_ Counter. Calculate the variable Row_Address from the start of the Machine Code given by the start (mem_start_row) plus the content of Add_Bus_Reg. Set the Col_Address. Set the cell Ins_Register to the content of the cell calculated.
Rem ' Also set-up Accumulator value in Shift register and Carry Shift_Acc.Value = Accumulator.Value Carry_in.Value = Carry_flag.Value	Set the Accumulator and Carry flag into cells held by the ALU Acc sheet. This preserves the integrity of the sheet when the values change in the Processor. These are only significant for ALU Accumulator instructions.

```
Rem '
Rem ' Other MC Count is Execute cycle
Rem '
Rem '
Rem ' Immediate Instructions
Rem '
   ElseIf Immediate.String = "Y" Then
      Program_Counter.Value = Program_Counter.Value + 1
      If MC_count.Value = 2 Then
         Add_Bus_Reg.Value = Program_Counter.Value
         Row_Address = mem_start_row.Value + Add_Bus_Reg.Value
         Col_Address = mem_start_col.Value
         If Load_ins.String = "Y" Then
            temp = Processor.getCellByPosition(Col_Address, Row_Address)
            Accumulator.Value = temp.value
            LDI_flag.setString("Y")
         ElseIf ALU.String = "Y" Then
            temp = Processor.getCellByPosition(Col_Address, Row_Address)
            ALU_Mem.Value = temp.value
            ALU_Acc.Value = Accumulator.Value
            ALU_Result.Value = ALU_out.Value
            If Add_flag.String = "Y" Then
               Carry_flag.Value = Carry_out_add.Value
            ElseIf Sub_flag.String = "Y" Then
               Carry_flag.Value = Carry_out_sub.Value
            End If
         End If
```

If MC count is not one the cycle is an Execute cycle. The Immediate flag is checked. If the flag is set the following processing occurs after which the macro ends.

The PC is incremented for each clock entry of the immediate instruction. As described before, the emulator increments the PC at the start of processing, so occurs here for MC count 2 and 3.

For MC count equals 2 the operand cell address is calculated and processing occurs separately for the LDI and ALU instructions.

For LDI the cell content is copied to the Accumulator. The LDI flag is set so that the PC is incremented on the next clock because no further entry here occurs for LDI (MC_count is equal to MC_cycles at this point).

For ALU the cell content is copied to the ALU_Mem cell. The Accumulator is copied to the ALU_Acc cell. The ALU Acc sheet provides the result for the appropriate arithmetic/logic instruction in ALU_out which is copied into ALU_Result.

For arithmetic instructions the appropriate output carry is copied to the Carry flag.

Rem
ElseIf MC_count.Value = 3 Then
If ALU.String = "Y" Then
Accumulator.Value = ALU_Result.Value
End If
End If
	For an immediate instruction of three Machine Cycles the code following the test here is processed when MC count equals three. This can only be an ALU instruction in this design (but is tested anyway) and writes the result of the ALU into the Accumulator.

Rem ' Rem ' Memory operation instructions Rem ' ElseIf Load_Address.String = "Y" Then If MC_count.Value = 2 Then Program_Counter.Value = Program_Counter.Value + 1 Add_Bus_Reg.Value = Program_Counter.Value Row_Address = mem_start_row.Value + Add_Bus_Reg.Value Col_Address = mem_start_col.Value temp = Processor.getCellByPosition(Col_Address, Row_Address) Address_Register.Value = temp.value	If MC count is not one and the instruction is not immediate, a check is made for an instruction operating memory. This includes load, store and ALU instructions. For MC count equals 2 the PC is incremented and the Address Register loaded with the operand value using the same address calculation seen in the above.

ElseIf MC_count.Value = 3 Then Program_Counter.Value = Program_Counter.Value + 1 If Indirect.String = "Y" Then Add_Bus_Reg.Value = Address_Register.Value ElseIf Load_ins.String = "Y" Then Add_Bus_Reg.Value = Address_Register.Value Row_Address = mem_start_row.Value + Add_Bus_Reg.Value Col_Address = mem_start_col.Value temp = Processor.getCellByPosition(Col_Address, Row_Address) Accumulator.Value = temp.value	For MC count equals 3 the PC is incremented. For indirect instructions the Add_Bus_Reg is set to the value in Address_Register. No further processing occurs for this cycle. For LDA the Accumulator is loaded with the value in memory using the same address calculation seen in the above.

ElseIf Store.String = "Y" Then Add_Bus_Reg.Value = Address_Register.Value Row_Address = mem_start_row.Value + Add_Bus_Reg.Value Col_Address = mem_start_col.Value temp = Processor.getCellByPosition(Col_Address, Row_Address) temp.value = Accumulator.Value	For STA the address calculation is as seen before but the copy is from Accumulator to memory.

ElseIf ALU.String = "Y" Then
Add_Bus_Reg.Value = Address_Register.Value
Row_Address = mem_start_row.Value + Add_Bus_Reg.Value
Col_Address = mem_start_col.Value
temp = Processor.getCellByPosition(Col_Address, Row_Address)
ALU_Mem.Value = temp.value
ALU_Acc.Value = Accumulator.Value
ALU_Result.Value = ALU_out.Value
If Add_flag.String = "Y" Then
Carry_flag.Value = Carry_out_add.Value
ElseIf Sub_flag.String = "Y" Then
Carry_flag.Value = Carry_out_sub.Value
End If
End If
	For ALU instructions the address calculation is as seen before and the data is loaded into ALU_Mem. The Accumulator is copied into ALU_Acc and the result from the ALU Acc sheet ALU_out into ALU_Result.
	
	For arithmetic instructions the appropriate output carry is copied to the Carry flag.

Rem
ElseIf MC_count.Value = 4 Then
If Indirect.String = "Y" Then
Row_Address = mem_start_row.Value + Add_Bus_Reg.Value
Col_Address = mem_start_col.Value
temp = Processor.getCellByPosition(Col_Address, Row_Address)
Address_Register.Value = temp.Value
ElseIf ALU.String = "Y" Then
Accumulator.Value = ALU_Result.Value
End If
	For MC count equals four the instruction is either indirect or an ALU operation.
	
	For indirect instructions the memory address to the data is calculated and loaded into the Address Register.
	
	For ALU instructions the result is copied to the accumulator.

Rem
ElseIf MC_count.Value = 5 Then
Add_Bus_Reg.Value = Address_Register.Value
Row_Address = mem_start_row.Value + Add_Bus_Reg.Value
Col_Address = mem_start_col.Value
If Load_ins.String = "Y" Then
temp = Processor.getCellByPosition(Col_Address, Row_Address)
Accumulator.Value = temp.Value
ElseIf Store.String = "Y" Then
temp = Processor.getCellByPosition(Col_Address, Row_Address)
temp.Value = Accumulator.Value
End If
End If
	For MC count equals five the instruction is an indirect memory operation.
	
	The address of the data is calculated. Data is loaded into the Accumulator, or stored in memory, according to a read or write instruction.

Rem '
Rem ' Miscellaneous instructions
Rem '
ElseIf Shift.String = "Y" Then
If MC_count.Value = 2 Then
Program_Counter.Value = Program_Counter.Value + 1
Accumulator.Value = Shift_out.Value
Carry_flag.Value = Carry_out_shift.Value
End If
	The Accumulator instructions are processed here. The PC is incremented and the output from the ALU Acc sheet copied into the Accumulator and Carry flag.

Rem
ElseIf Clear_Carry.String = "Y" Then
If MC_count.Value = 2 Then
Program_Counter.Value = Program_Counter.Value + 1
Carry_flag.Value = 0
End If
	The Clear Carry instruction forces the Carry flag to zero.

Rem
ElseIf Set_Carry.String = "Y" Then
If MC_count.Value = 2 Then
Program_Counter.Value = Program_Counter.Value + 1
Carry_flag.Value = 1
End If
	The Set Carry instruction forces the Carry flag to one.

Rem ' Rem ' Branch instructions Rem ' ElseIf Branch.String = "Y" Then If MC_count.Value = 2 Then Program_Counter.Value = Program_Counter.Value + 1 Add_Bus_Reg.Value = Program_Counter.Value Row_Address = mem_start_row.Value + Add_Bus_Reg.Value Col_Address = mem_start_col.Value temp = Processor.getCellByPosition(Col_Address, Row_Address) Address_Register.Value = temp.Value	Branch instruction execute cycle processing occurs here. For MC count equals two the Address Register is loaded with the instruction operand in the same way as seen before.

Rem
ElseIf MC_count.Value = 3 Then
Program_Counter.Value = Program_Counter.Value + 1
If BrDecision.Value = 1 Then
Program_Counter.Value = Address_Register.Value
End If
End If
Rem
End If
Rem
End Sub
	For MC count equals three the Branch condition result is checked and the PC updated by the Address Register if Gate H is opened.
	
	End of macro

Sub Reset() Doc = ThisComponent Processor = Doc.Sheets.getByName("Processor") ALU_Shift = Doc.Sheets.getByName("ALU Acc") clock_cycle = Processor.getCellrangeByName("clock_cycle") MC_count = Processor.getCellrangeByName("MC_count") Program_Counter = Processor.getCellrangeByName("Program_Counter") Add_Bus_Reg = Processor.getCellrangeByName("Add_Bus_Reg") Ins_Register = Processor.getCellrangeByName("Ins_Register") Accumulator = Processor.getCellrangeByName("Accumulator") ALU_Result = Processor.getCellrangeByName("ALU_Result") Address_Register = Processor.getCellrangeByName("Address_Register") Carry_flag = Processor.getCellrangeByName("Carry_flag") ALU_Acc = Processor.getCellrangeByName("ALU_Acc") ALU_Mem = Processor.getCellrangeByName("ALU_Mem") Fetch = Processor.getCellrangeByName("Fetch") Execute = Processor.getCellrangeByName("Execute") LDI_flag = Processor.getCellrangeByName("LDI_flag") Carry_in = ALU_Shift.getCellrangeByName("Carry_in")	Clicking the “Reset” button runs this program. In OpenOffice Basic the objects used in the program have to be explicitly defined. The document, sheets and cells are explicitly defined here for the macro.

clock_cycle.Value = 0 MC_count.Value = 0 Program_Counter.Value = 0 Add_Bus_Reg.Value = 0 Ins_Register.Value = 0 Accumulator.Value = 0 ALU_Result.Value = 0 Address_Register.Value = 0 Carry_flag.Value = 0 ALU_Acc.Value = 0 ALU_Mem.Value = 0 Fetch.setString("N") Execute.setString("N") LDI_flag.setString("N") Carry_in.Value = 0 End Sub	The named cells are set to the value shown (0 or N).

sub Assembler rem ---------- rem define variables dim document as object dim dispatcher as object rem ---------- rem get access to the document document = ThisComponent.CurrentController.Frame dispatcher = createUnoService("com.sun.star.frame.DispatchHelper") rem ---------- dim args1(0) as new com.sun.star.beans.PropertyValue args1(0).Name = "Nr" args1(0).Value = 3 dispatcher.executeDispatch(document, ".uno:JumpToTable", "", 0, args1()) rem ---------- dim args2(0) as new com.sun.star.beans.PropertyValue args2(0).Name = "ToPoint" args2(0).Value = "$C$2" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args2())	This macro copies the content of the editable cells on the Source sheet into hidden cells on the Assembler sheet. Only the values are copied. The hidden cells are visible in the copy shown in the panel. The cells are deselected and the active sheet on macro end is the Assembler. The macro was constructed using macro recording.

```
rem ----------------------------------------------------------------------
dim args3(0) as new com.sun.star.beans.PropertyValue
args3(0).Name = "ToPoint"
args3(0).Value = "$C$2:$E$129"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args3())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())

rem ----------------------------------------------------------------------
dim args5(0) as new com.sun.star.beans.PropertyValue
args5(0).Name = "Nr"
args5(0).Value = 2

dispatcher.executeDispatch(document, ".uno:JumpToTable", "", 0, args5())

rem ----------------------------------------------------------------------
dim args6(0) as new com.sun.star.beans.PropertyValue
args6(0).Name = "ToPoint"
args6(0).Value = "$AJ$2"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args6())
```

```
rem ----------------------------------------------------------------------
dim args7(5) as new com.sun.star.beans.PropertyValue
args7(0).Name = "Flags"
args7(0).Value = "SVD"
args7(1).Name = "FormulaCommand"
args7(1).Value = 0
args7(2).Name = "SkipEmptyCells"
args7(2).Value = false
args7(3).Name = "Transpose"
args7(3).Value = false
args7(4).Name = "AsLink"
args7(4).Value = false
args7(5).Name = "MoveMode"
args7(5).Value = 4

dispatcher.executeDispatch(document, ".uno:InsertContents", "", 0, args7())

rem ----------------------------------------------------------------------
dim args8(0) as new com.sun.star.beans.PropertyValue
args8(0).Name = "ToPoint"
args8(0).Value = "$A$4"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args8())


end sub
```

sub Load_Memory rem ---------- rem define variables dim document as object dim dispatcher as object rem ---------- rem get access to the document document = ThisComponent.CurrentController.Frame dispatcher = createUnoService("com.sun.star.frame.DispatchHelper") rem ---------- dim args1(0) as new com.sun.star.beans.PropertyValue args1(0).Name = "Nr" args1(0).Value = 2 dispatcher.executeDispatch(document, ".uno:JumpToTable", "", 0, args1()) rem ---------- dim args2(0) as new com.sun.star.beans.PropertyValue args2(0).Name = "ToPoint" args2(0).Value = "$AF$3" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args2())	This macro copies the content of the output cells on the Assembler sheet into hidden cells on the Processor sheet. Only the values are copied. The hidden cells are visible in the copy shown in the memory panel according to the selected format (hex, decimal or disassemble). The cells are deselected and the active sheet on macro end is the Processor. The macro was constructed using macro recording. LOC_count was added to limit the disassemble view to the instruction codes and not subsequent data.

```
rem ----------------------------------------------------------------------
dim args3(0) as new com.sun.star.beans.PropertyValue
args3(0).Name = "ToPoint"
args3(0).Value = "$AF$3:$AF$258"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args3())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())

rem ----------------------------------------------------------------------
dim args5(0) as new com.sun.star.beans.PropertyValue
args5(0).Name = "Nr"
args5(0).Value = 1

dispatcher.executeDispatch(document, ".uno:JumpToTable", "", 0, args5())

rem ----------------------------------------------------------------------
dim args6(0) as new com.sun.star.beans.PropertyValue
args6(0).Name = "ToPoint"
args6(0).Value = "$BA$25"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args6())
```

```
rem ----------------------------------------------------------------------
dim args7(5) as new com.sun.star.beans.PropertyValue
args7(0).Name = "Flags"
args7(0).Value = "SVD"
args7(1).Name = "FormulaCommand"
args7(1).Value = 0
args7(2).Name = "SkipEmptyCells"
args7(2).Value = false
args7(3).Name = "Transpose"
args7(3).Value = false
args7(4).Name = "AsLink"
args7(4).Value = false
args7(5).Name = "MoveMode"
args7(5).Value = 4

dispatcher.executeDispatch(document, ".uno:InsertContents", "", 0, args7())

Doc = ThisComponent

Processor = Doc.Sheets.getByName("Processor")
temp = Doc.Sheets.getByName("Assembler")

LOC_count = Processor.getCellrangeByName("LOC_count")
LOC_sum = temp.getCellrangeByName("LOC_sum")

LOC_count.Value = LOC_sum.Value
```

rem ---------- dim args8(0) as new com.sun.star.beans.PropertyValue args8(0).Name = "ToPoint" args8(0).Value = "$B$18" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args8()) end sub	

sub Clear_Memory rem ---------- rem define variables dim document as object dim dispatcher as object rem ---------- rem get access to the document document = ThisComponent.CurrentController.Frame dispatcher = createUnoService("com.sun.star.frame.DispatchHelper") rem ---------- dim args1(0) as new com.sun.star.beans.PropertyValue args1(0).Name = "ToPoint" args1(0).Value = "$AC$25" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args1())	This macro copies the content of a range of cells set to zero on the Processor sheet into hidden memory cells on the Processor sheet. Only the values are copied. The hidden cells are visible in the copy shown in the memory panel according to the selected format (hex, decimal or disassemble). The cells are deselected and the active sheet on macro end is the Processor. The macro was constructed using macro recording. LOC_count was added to limit the disassemble view to the Instruction Codes and not subsequent data. It is set to zero.

```
rem ----------------------------------------------------------------------
dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).Name = "ToPoint"
args2(0).Value = "$AC$25:$AC$280"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args2())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())

rem ----------------------------------------------------------------------
dim args4(0) as new com.sun.star.beans.PropertyValue
args4(0).Name = "ToPoint"
args4(0).Value = "$BA$25"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args4())
```

```
rem ----------------------------------------------------------------------
dim args5(5) as new com.sun.star.beans.PropertyValue
args5(0).Name = "Flags"
args5(0).Value = "SVD"
args5(1).Name = "FormulaCommand"
args5(1).Value = 0
args5(2).Name = "SkipEmptyCells"
args5(2).Value = false
args5(3).Name = "Transpose"
args5(3).Value = false
args5(4).Name = "AsLink"
args5(4).Value = false
args5(5).Name = "MoveMode"
args5(5).Value = 4

dispatcher.executeDispatch(document, ".uno:InsertContents", "", 0, args5())

Doc = ThisComponent

Processor = Doc.Sheets.getByName("Processor")

LOC_count = Processor.getCellrangeByName("LOC_count")

LOC_count.Value = 0
```

rem -- dim args6(0) as new com.sun.star.beans.PropertyValue args6(0).Name = "ToPoint" args6(0).Value = "$B$18" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args6()) end sub	

sub Clear_Code rem ---------- rem define variables dim document as object dim dispatcher as object rem ---------- rem get access to the document document = ThisComponent.CurrentController.Frame dispatcher = createUnoService("com.sun.star.frame.DispatchHelper") rem ---------- dim args1(0) as new com.sun.star.beans.PropertyValue args1(0).Name = "ToPoint" args1(0).Value = "$C$2" dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args1())	This macro copies the content of a range of cells set to blank on the Source sheet into editable panel on the Source sheet. Only the values are copied. The cells are deselected. The macro was constructed using macro recording.

```
rem ----------------------------------------------------------------------
dim args2(0) as new com.sun.star.beans.PropertyValue
args2(0).Name = "ToPoint"
args2(0).Value = "$C$2:$E$129"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args2())

rem ----------------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:ClearContents", "", 0, Array())

rem ----------------------------------------------------------------------
dim args4(0) as new com.sun.star.beans.PropertyValue
args4(0).Name = "ToPoint"
args4(0).Value = "$A$9"

dispatcher.executeDispatch(document, ".uno:GoToCell", "", 0, args4())

end sub
```

Sub Select_Hex() Doc = ThisComponent Processor = Doc.Sheets.getByName("Processor") DHDSwitch = Processor.getCellrangeByName("DHDSwitch") DHDSwitch.Value = 1 End Sub	The three macros set the value in cell DHDSwitch as showm. DHDSwitch is ised in IF statements to control the memory view.
Sub Select_Dec() Doc = ThisComponent Processor = Doc.Sheets.getByName("Processor") DHDSwitch = Processor.getCellrangeByName("DHDSwitch") DHDSwitch.Value = 0 End Sub	
Sub Select_Dis() Doc = ThisComponent Processor = Doc.Sheets.getByName("Processor") DHDSwitch = Processor.getCellrangeByName("DHDSwitch") DHDSwitch.Value = 2 End Sub	

Sub Create_file Dim FileNo As Integer Dim Filename As String Dim MyByte As Byte Dim path As String Doc = ThisComponent Mysheet = Doc.Sheets.getByName("Assembler") GlobalScope.BasicLibraries.loadLibrary("Tools") path = DirectoryNameoutofPath(Doc.getURL(),"/") Filename = path & "/Firmware.bin" FileNo = FreeFile Open Filename For Binary As #FileNo For i = 2 to 257 MyByte = CByte(Mysheet.getCellByPosition(31,i).Value) Put #FileNo, , MyByte Next i Close #FileNo MsgBox "Done!!" End Sub	This macro creates a binary file for use by the simulator (Book 3). The macro opens or creates and opens the file “Firmware.bin” in the same directory as the emulator. The binary file is defined by the 256 bytes created from the 256 cells running below the cell “Assemb_output” explicitly referenced by cell position. Once defined, the file is saved and closed. The message box provides positive feedback to the user that the task has been performed successfully.