KCPSM6 Instruction Set

```
aaa : 12-bit address 000 to FFF
 kk : 8-bit constant 00 to FF
 pp : 8-bit port ID 00 to FF
  p : 4-bit port ID 0 to F
 ss : 8-bit scratch pad location 00 to FF
  x : Register within bank s0 to sF
  y : Register within bank s0 to sF
```

Page	Opcode	Instruction
	Register loading	
55	00xy0	LOAD sX, sY
55	01xkk	LOAD sX, kk
71	16xy0	STAR sX, sY
	Logical	
56	02xy0	AND sX, sY
56	03xkk	AND sX, kk
57	04xy0	OR sX, sY
57	05xkk	OR sX, kk
58	06xy0	XOR sX, sY
58	07xkk	XOR sX, kk
	Arithmetic	
59	10xy0	ADD sX, sY
59	11xkk	ADD sX, kk
60	12xy0	ADDCY sX, sY
60	13xkk	ADDCY sX, kk
61	18xy0	SUB sX, sY
61	19xkk	SUB sX, kk
62	1Axy0	SUBCY sX, sY
62	1Bxkk	SUBCY sX, kk
	Test and Compare	
63	0Cxy0	TEST sX, sY
63	0Dxkk	TEST sX, kk
64	0Exy0	TESTCY sX, sY
64	0Fxkk	TESTCY sX, kk
65	1Cxy0	COMPARE sX, sY
65	1Dxkk	COMPARE sX, kk
66	1Exy0	COMPARECY sX, sY
66	1Fxkk	COMPARECY sX, kk

Page	Opcode	Instruction
	Shift and Rotate	
67	14x06	SL0 sX
67	14x07	SL1 sX
67	14x04	SLX sX
67	14x00	SLA sX
67	14x02	RL sX
68	14x0E	SR0 sX
68	14x0F	SR1 sX
68	14x0A	SRX sX
68	14x08	SRA sX
68	14x0C	RR sX
	Register Bank Selection	
70	37000	REGBANK A
70	37001	REGBANK B
	Input and Output	
73	08xy0	INPUT sX, (sY)
73	09xpp	INPUT sX, pp
74	2Cxy0	OUTPUT sX,(sY)
74	2Dxpp	OUTPUT sX, pp
78	2Bkkp	OUTPUTK kk, p
	Scratch Pad Memory	
	(64, 128 or 256 bytes)	
81	2Exy0	STORE sX,(sY)
81	2Fxss	STORE sX, ss
82	0Axy0	FETCH sX, (sY)
82	0Bxss	FETCH sX, ss

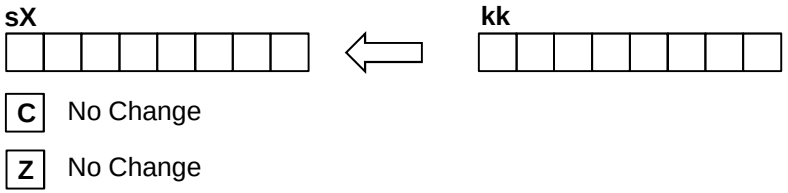
Page	Opcode	Instruction
	Interrupt Handling	
83	28000	DISABLE INTERRUPT
83	28001	ENABLE INTERRUPT
84	29000	RETURNI DISABLE
84	29001	RETURNI ENABLE
	Jump	
87	22aaa	JUMP aaa
88	32aaa	JUMP Z, aaa
88	36aaa	JUMP NZ, aaa
88	3Aaaa	JUMP C, aaa
88	3Eaaa	JUMP NC, aaa
89	26xy0	JUMP@ (sX, sY)
	Subroutines	
92	20aaa	CALL aaa
93	30aaa	CALL Z, aaa
93	34aaa	CALL NZ, aaa
93	38aaa	CALL C, aaa
93	3Caaa	CALL NC, aaa
94	24xy0	CALL@ (sX, sY)
96	25000	RETURN
97	31000	RETURN Z
97	35000	RETURN NZ
97	39000	RETURN C
97	3D000	RETURN NC
98	21xkk	LOAD&RETURN sX, kk
	Version Control	
100	14x80	HWBUILD sX

XILINX

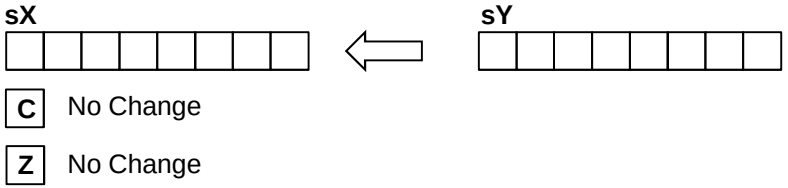
LOAD sX, kk
LOAD sX, sY

The ‘LOAD’ instructions provides a simple way to define the contents of any register (sX). The value loaded into a register can a fixed value (kk) or the value contained in any register (sY) can be copied. The states of the zero flag (Z) and the carry flag (C) will be unaffected.

LOAD sX, kk sX = kk



LOAD sX, sY sX = kY



Examples

```
LOAD sA, 8E
LOAD s4, 42'd
LOAD s6, "k"
LOAD s7, sA
```

The KCPSM6 assembler enables constant values in all instructions that require them to be defined in hexadecimal (default), decimal or using a single character which is converted to its ASCII equivalent value.

After this example has executed both ‘s7’ and ‘sA’ will contain 8E hex. ‘sA’ will contain 2A hex and ‘s6’ will contain 77 hex.

Hint - Loading a register with itself has no effect other than taking 2 clock cycles but this can be useful way to create a known delay.

Notes

‘sX’ and ‘sY’ define any of the 16 registers in the range ‘s0’ through to ‘sF’ in the active register bank. Please see the ‘Using Register Banks’ section to see how to switch between the ‘A’ and ‘B’ register banks and techniques for copying values from registers in one bank to registers in the other.

The instruction op-code 00000 hex was specifically assigned to be the instruction ‘LOAD s0, s0’. In this way the default value (zero) of any unused program memory contents will have the minimum effect should an improper program result in these undefined locations being executed.

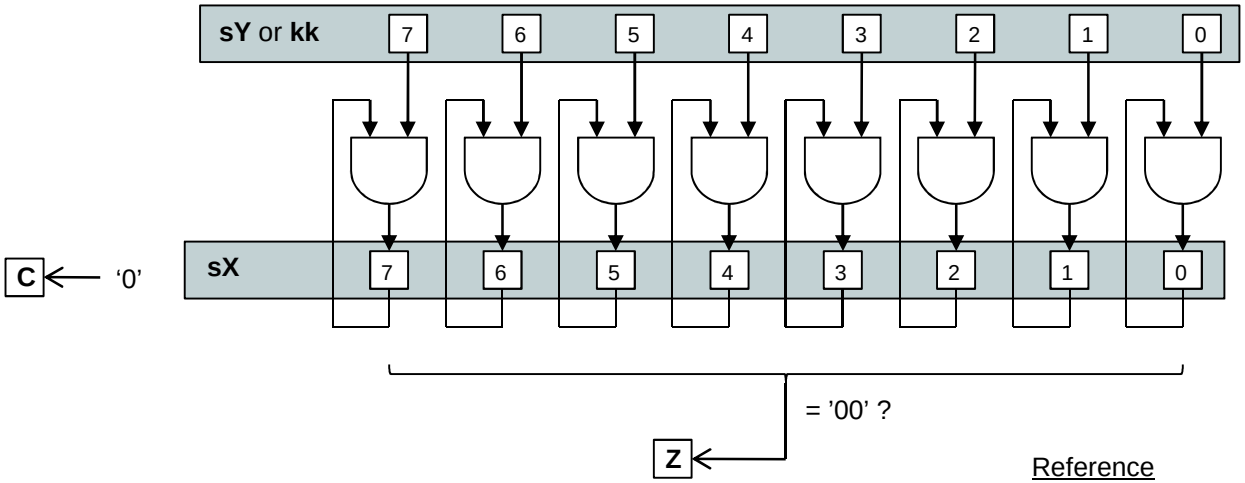
AND sX, kk
AND sX, sY

The ‘AND’ instructions perform the bit-wise logical AND operation.

The first operand must specify a register ‘sX’ whose value provides one input to the AND operation and in to which the result is returned.
The second operand defines the second input to the AND operation and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if all 8-bits of the result returned to ‘sX’ are zero.
The carry flag (C) will be cleared (C=0) in all cases.

AND sX, kk sX = sX AND kk

AND sX, sY sX = sX AND sY



Examples

```
LOAD sA, CA
AND sA, 53
```

sA = 42, Z=0, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      53 = 0 1 0 1 0 0 1 1
CA AND 53 = 0 1 0 0 0 0 1 0 = 42
```

```
LOAD sA, CA
AND sA, 14
```

sA = 00, Z=1, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      14 = 0 0 0 1 0 1 0 0
CA AND 14 = 0 0 0 0 0 0 0 0 = 00
```

```
AND s0, 0F
```

Hint – This instruction will clear the upper nibble of ‘s0’ and preserve the lower nibble. This could be used to convert the ASCII characters ‘0’ to ‘9’ (30 to 39 hex) into their equivalent numerical values (00 to 09 hex).

Reference

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

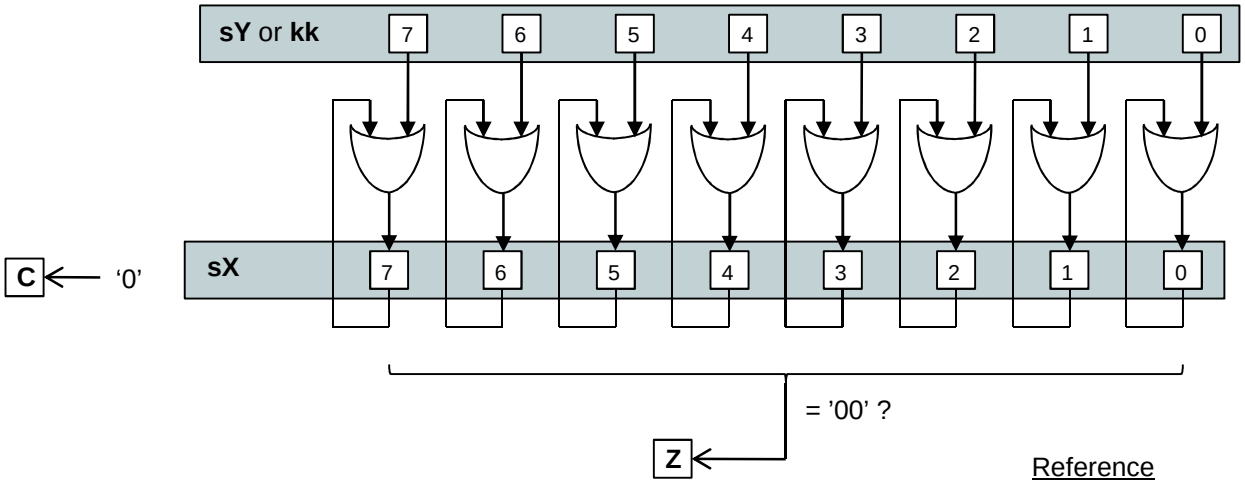
OR sX, kk
OR sX, sY

The ‘OR’ instructions perform the bit-wise logical OR operation.

The first operand must specify a register ‘sX’ whose value provides one input to the OR operation and in to which the result is returned.
The second operand defines the second input to the OR operation and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if all 8-bits of the result returned to ‘sX’ are zero.
The carry flag (C) will be cleared (C=0) in all cases.

OR sX, kk sX = sX OR kk

OR sX, sY sX = sX OR sY



Examples

```
LOAD sA, CA
OR sA, 53
```

sA = DB, Z=0, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      53 = 0 1 0 1 0 0 1 1
CA OR 53 = 1 1 0 1 1 0 1 1 = DB
```

```
LOAD sA, CA
OR sA, 14
```

sA = DE, Z=1, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      14 = 0 0 0 1 0 1 0 0
CA OR 14 = 1 1 0 1 1 1 1 0 = DE
```

```
OR s0, 30
```

Hint – This instruction will set bits4 and bit5 of ‘s0’ and preserve all other bits. This could be used to convert the numerical values 00 to 09 hex into their ASCII equivalent characters ‘0’ to ‘9’ (30 to 39 hex).

Reference

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

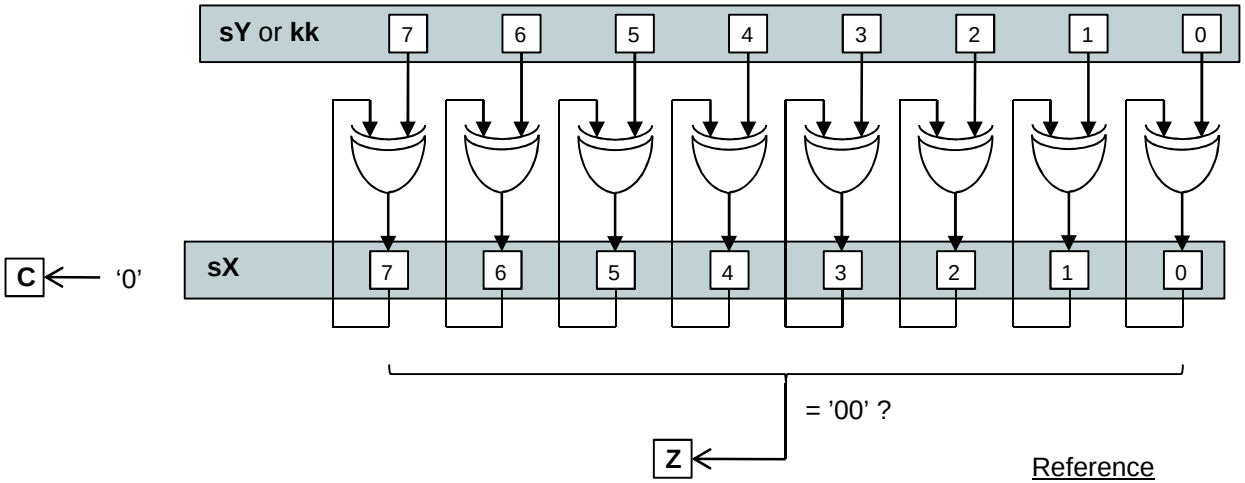
XOR sX, kk
XOR sX, sY

The ‘XOR’ instructions perform the bit-wise logical exclusive-OR operation.

The first operand must specify a register ‘sX’ whose value provides one input to the XOR operation and in to which the result is returned.
The second operand defines the second input to the XOR operation and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if all 8-bits of the result returned to ‘sX’ are zero.
The carry flag (C) will be cleared (C=0) in all cases.

XOR sX, kk sX = sX XOR kk

XOR sX, sY sX = sX XOR sY



Examples

```
LOAD sA, CA
XOR sA, 53
```

sA = 99, Z=0, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      53 = 0 1 0 1 0 0 1 1
CA XOR 53 = 1 0 0 1 1 0 0 1 = 99
```

```
LOAD sA, CA
XOR sA, 14
```

sA = DE, Z=1, C=0.

```
      CA = 1 1 0 0 1 0 1 0
      14 = 0 0 0 1 0 1 0 0
CA XOR 14 = 1 1 0 1 1 1 1 0 = DE
```

```
XOR s0, 01
OUTPUT s0, 08
XOR s0, 01
OUTPUT s0, 08
```

Hint – The XOR instruction can be used to toggle the state of bits within a register. In this example the least significant bit of ‘s0’ is twice toggled and output to a port. Assuming the LSB was ‘0’ to begin with then this will have generated a positive (‘1’) pulse on the LSB of the output port whilst all other bits remained unaffected.

Reference

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

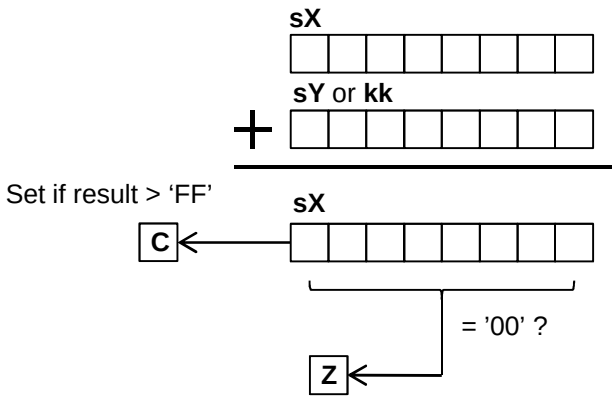
ADD sX, kk
ADD sX, sY

The ‘ADD’ instructions perform the arithmetic addition of two 8-bit values and set the carry and zero flags according to the result.

The first operand must specify a register ‘sX’ whose value provides one input to the addition function and in to which the result is returned.
The second operand defines the second input to the addition and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the 8-bit result returned to ‘sX’ is zero.
The carry flag (C) will be set if the addition results in an overflow.

ADD sX, kk sX = sX + kk

ADD sX, sY sX = sX + sY



Examples

```
LOAD sA, 8E
ADD sA, 43
```
8E + 43 = D1 sA = D1 which is not zero (Z=0) and with no overflow (C=0).

```
LOAD sA, 8E
ADD sA, sA
```
8E + 8E = 11C sA = 1C which is not zero (Z=0) but there was an overflow (C=0).

```
LOAD sA, 8E
ADD sA, 72
```
8E + 72 = 100 sA = 00 which is zero (Z=1) but there was also an overflow that made this happen (C=1).

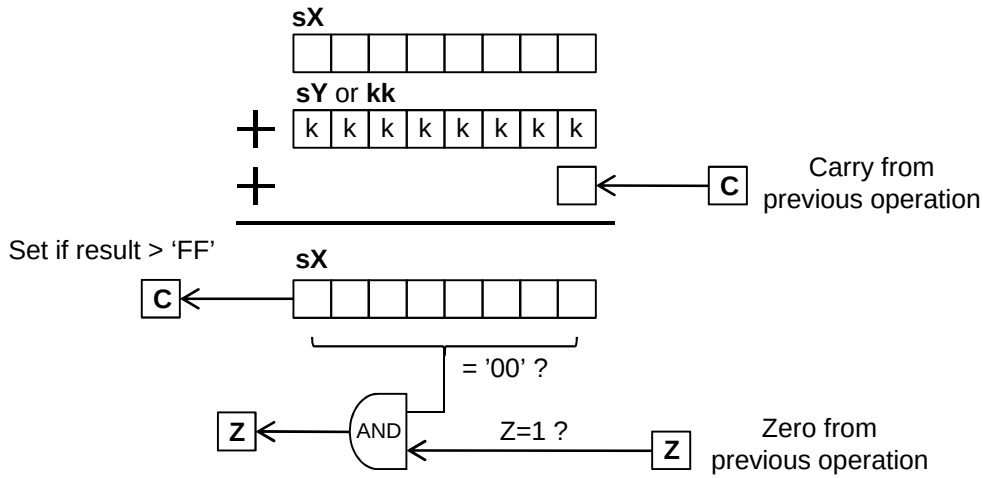
ADDCY sX, kk
ADDCY sX, sY

The ‘ADDCY’ instructions are primarily intended as an extension to the basic ‘ADD’ instructions in order to support arithmetic addition of values more than 8-bits. The key difference from the ADD instructions is that the zero and carry flags are also used as inputs to the addition function and these can influence both the 8-bit result and the new states of the flags. Although each register only contains an 8-bit value, any combination of registers can be used to hold larger values segmented into bytes. For example a 32-bit value can be held in a 4 registers. Although there is no restriction on which registers, and no formal way of describing the assignment, it is common practice to assign adjacent registers and refer to them as a ‘register set’ such as [sD, sC, sB, sA].

The first operand must specify a register ‘sX’ whose value provides one input to the addition function and in to which the result is returned.
The second operand defines the second input to the addition and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the 8-bit result returned to ‘sX’ is zero *and* the zero flag was set prior to the ADDCY instruction.
The carry flag (C) will be set if the addition results in an overflow.

ADDCY sX, kk sX = sX + kk + C

ADDCY sX, sY sX = sX + sY + C



Examples

The key observation to make as illustrated by these examples is that carry and zero flags reflect the entire result of a multi-byte addition. In particular, the zero flag is only set if the complete multi-byte result is zero and is not just based on the 8-bit result of the final ADDCY operation.

```
LOAD sA, 7B
LOAD sB, A2
ADD sA, 85
ADDCY sB, 5D
```

[sB, sA] = A2 7B
+ 5E 1A = 10095

7B + 1A = 95 sA = 95, Z=0, C=0.
A2 + 5E + 0 = 100 sB = 00, Z=0, C=1.

```
LOAD sA, 7B
LOAD sB, A2
ADD sA, 85
ADDCY sB, 5D
```

[sB, sA] = A2 7B
+ 5D 85 = 10000

7B + 85 = 100 sA = 00, Z=1, C=1.
A2 + 5D + 1 = 100 sB = 00, Z=1, C=1.

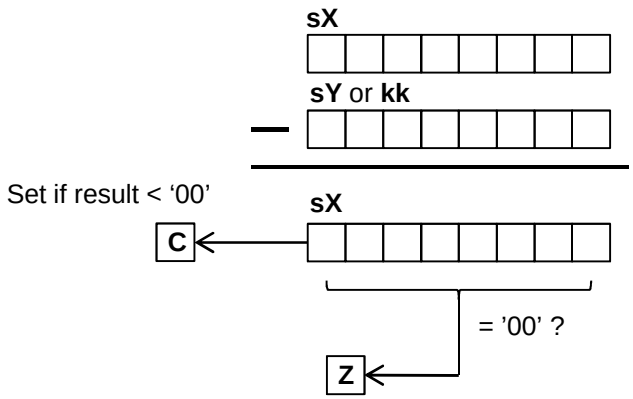
SUB sX, kk
SUB sX, sY

The ‘SUB’ instructions perform the arithmetic subtraction of two 8-bit values and set the carry and zero flags according to the result.

The first operand must specify a register ‘sX’ from which the second operand will be subtracted and to which the result is returned.
The second operand defines the value to be subtracted from the first operand and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the 8-bit result returned to ‘sX’ is zero.
The carry flag (C) will be set if the result of the subtraction is negative. Hence the carry flag represents an underflow or a ‘borrow’ to complete the operation.

SUB sX, kk sX = sX - kk

SUB sX, sY sX = sX - sY



Examples

```
LOAD sA, 8E
SUB sA, 43
```

8E - 43 = 4B sA = 4B which is not zero (Z=0) and with no underflow (C=0).

```
LOAD sA, 8E
ADD sA, sA
```

8E - 8E = 00 sA = 00 which is zero (Z=1) but there was no underflow (C=0).

```
LOAD sA, 8E
SUB sA, B5
```

8E – B5 = 1D9 sA = D9 which is not zero (Z=0) but there was an underflow (C=1).
This is equivalent to 142 – 181 = -39 where D9 hex is the twos complement representation of -39.
However, it is the users responsibility to implement and interpret signed arithmetic values and operations.

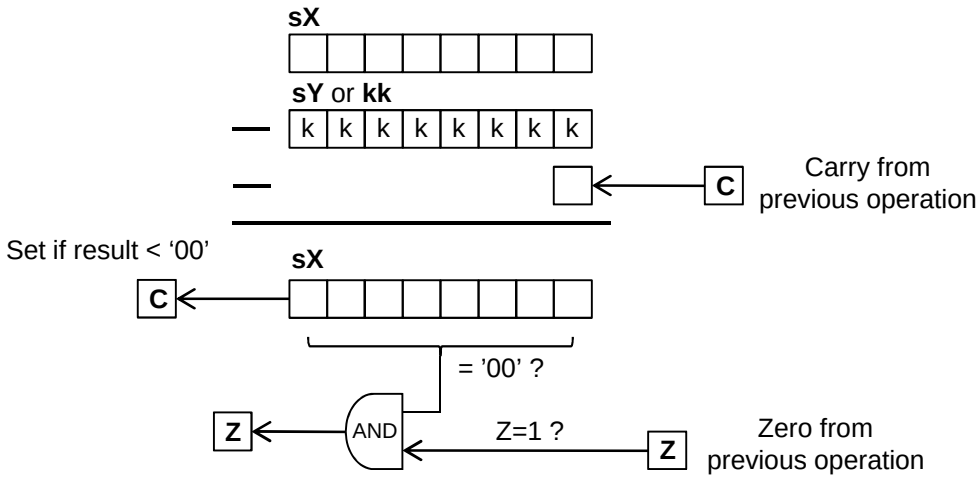
SUBCY sX, kk
SUBCY sX, sY

The ‘SUBCY’ instructions are primarily intended as an extension to the basic ‘SUB’ instructions in order to support arithmetic subtraction of values more than 8-bits. The key difference from the SUB instructions is that the zero and carry flags are also used as inputs to the subtraction function and these can influence both the 8-bit result and the new states of the flags. Although each register only contains an 8-bit value, any combination of registers can be used to hold larger values segmented into bytes. For example a 32-bit value can be held in a 4 registers. Although there is no restriction on which registers, and no formal way of describing the assignment, it is common practice to assign adjacent registers and refer to them as a ‘register set’ such as [sD, sC, sB, sA].

The first operand must specify a register ‘sX’ from which the second operand and carry flag will be subtracted and to which the result is returned.
The second operand defines the value to be subtracted from the first operand and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the 8-bit result returned to ‘sX’ is zero *and* the zero flag was set prior to the SUBCY instruction.
The carry flag (C) will be set if the result of the subtraction is negative. Hence the carry flag represents an underflow or a ‘borrow’ to complete the operation.

SUBCY sX, kk sX = sX - kk - C

SUBCY sX, sY sX = sX - sY - C



Examples

The key observation to make as illustrated by these examples is that carry and zero flags reflect the entire result of a multi-byte subtraction. In particular, the zero flag is only set if the complete multi-byte result is zero and is not just based on the 8-bit result of the final SUBCY operation.

```
LOAD sA, 7B
LOAD sB, A2
SUB sA, B9
SUBCY sB, A1
```

[sB, sA] = A2 7B
- A1 B9 = 00C2

7B – B9 = (-)C2 sA = C2, Z=0, C=1.
A2 – A1 - 1 = 00 sB = 00, Z=0, C=0.

```
LOAD sA, 7B
LOAD sB, A2
SUB sA, sA
SUBCY sB, sB
```

[sB, sA] = A2 7B
- A2 7B = 0000

7B – 7B = 00 sA = 00, Z=1, C=0.
A2 – A2 - 0 = 00 sB = 00, Z=1, C=0.

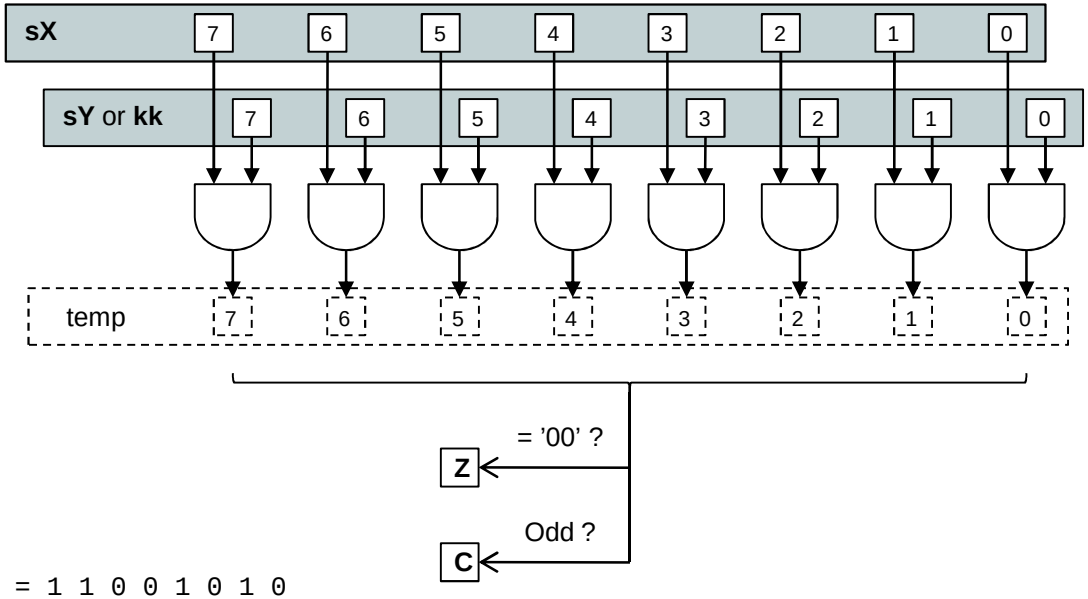
TEST sX, kk
TEST sX, sY

The ‘TEST’ instructions are similar to the ‘AND’ instructions in that a bit-wise logical AND operation is performed. However, the actual result is discarded and only the flags are updates to reflect the temporary 8-bit. The ‘TEST’ instruction also reports then exclusive-OR of the temporary result which can be used to compute the ‘odd parity’ of a value.

The first operand must specify a register ‘sX’ whose value provides one input to the AND operation (sX will not be effected by the operation).
The second operand defines the second input to the AND operation and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if all 8-bits of the temporary result are zero.
The carry flag (C) will be set if the temporary result contains an odd number of bits set to ‘1’ (the exclusive-OR of the 8-bit temporary result).

TEST sX, kk temp = sX AND kk

TEST sX, sY temp = sX AND sY



Hints

It is typical to think of ‘sX’ containing the information to be tested and for ‘sY’ or ‘kk’ to be acting as a bit-mask to select only those bits to be tested.

To test a single bit the value of ‘kk’ is best described using a binary format such as 00100000’b that will text bit 5 (equivalent to 20 hex). The ‘C’ flag will be set if the corresponding bit in ‘sX’ is ‘1’ and the ‘Z’ flag will be set if the tested bit is ‘0’.

Use a mask value of kk = FF to compute the odd parity of the whole byte contained in ‘sX’.

Examples

```
LOAD sA, CA
TEST sA, 01000000’b
```

Z=0, C=1 (odd).

```
      CA = 1 1 0 0 1 0 1 0
      40 = 0 1 0 0 0 0 0 0
CA AND 40 = 0 1 0 0 0 0 0 0 = 40
```

```
LOAD sA, 51
TEST sA, FF
```

Z=0, C=1.
Parity is odd.

```
      51 = 0 1 0 1 0 0 0 1
      FF = 1 1 1 1 1 1 1 1
51 AND FF = 0 1 0 1 0 0 0 1 = 51
```

Hint – The ‘SLA’ and ‘SRA’ shift instructions and the ADDCY and SUBCY instructions can all be used to move the value of the carry flag into a register.

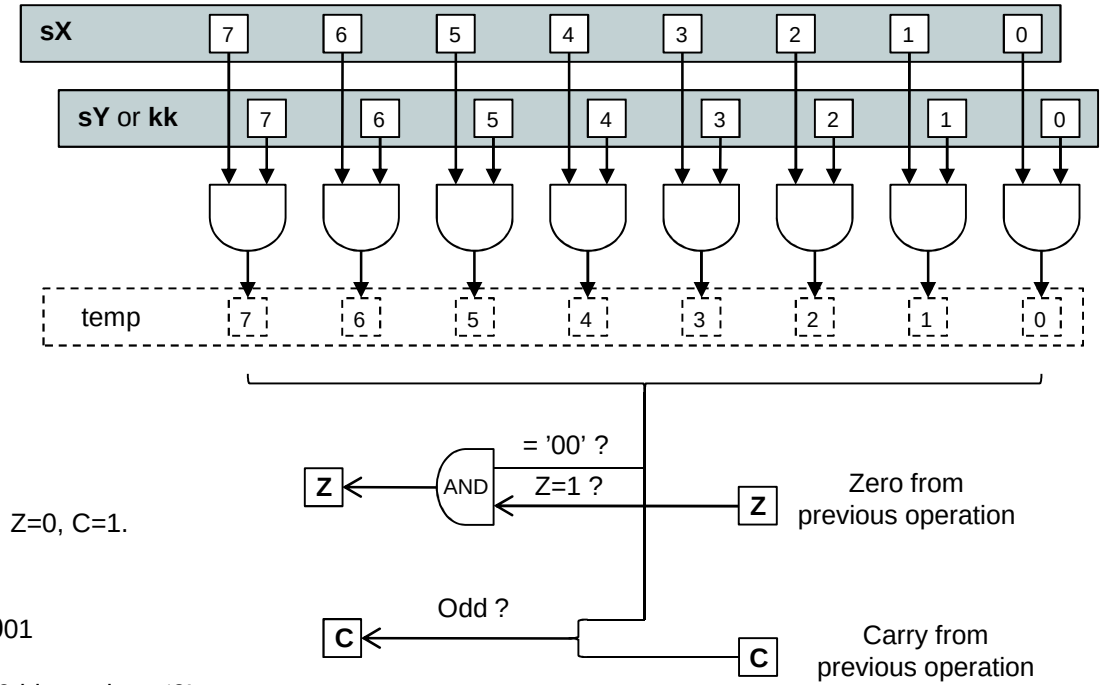
TESTCY sX, kk
TESTCY sX, sY

The ‘TESTCY’ instructions are primarily intended as an extension to the basic ‘TEST’ instructions in order to support testing and odd parity calculation of values more than 8-bits. The key difference from the TEST instructions is that the zero and carry flags are also used as inputs that can influence the new states of the flags. Although each register only contains an 8-bit value, any combination of registers can be used to hold larger values segmented into bytes. For example a 32-bit value can be held in a 4 registers. Although there is no restriction on which registers, and no formal way of describing the assignment, it is common practice to assign adjacent registers and refer to them as a ‘register set’ such as [sD, sC, sB, sA].

The first operand must specify a register ‘sX’ whose value provides one input to the AND operation (sX will not be effected by the operation).
The second operand defines the second input to the AND operation and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if all 8-bits of the temporary result are zero *and* the zero flag was set prior to the TESTCY instruction.
The carry flag (C) will be set if the temporary result together with the previous state of the carry flag contains an odd number of bits set to ‘1’.

TEST sX, kk temp = sX AND kk

TEST sX, sY temp = sX AND sY



The meaning of the ‘C’ and ‘Z’ flags are the same following a TEST and TESTCY combination of instructions used to test and compute the odd parity of a multi-byte value as they are after a single 8-bit TEST operation.

Examples

```
LOAD sA, CA
LOAD sB, 52
TEST sA, FF
TESTCY sB, FF
```

[sB, sA] = 11001010 0101001

7 bits in total are ‘1’ so parity is odd. Z=0, C=1.

```
LOAD sA, CA
LOAD sB, 52
TEST sA, 00000100’b
TESTCY sB, 00100000’b
```

[sB, sA] = 11**0**01010 0101**0**01

Both bit13 and bit3 of the 16-bit word are ‘0’.
Z=1, C=0 (even).

COMPARE sX, kk
COMPARE sX, sY

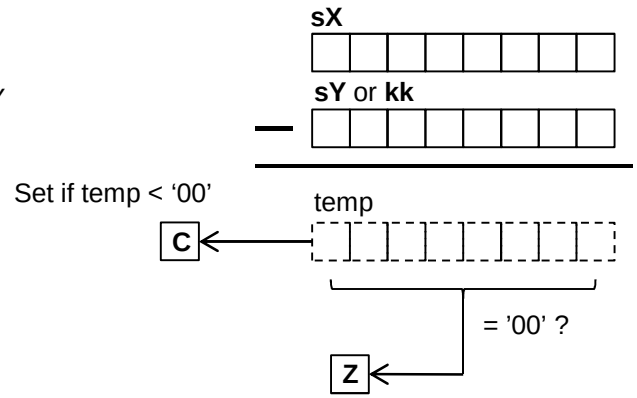
The ‘COMPARE’ instructions perform the arithmetic subtraction of two 8-bit values but the actual result is discarded and only the carry and zero flags are updated according to the temporary result.

The first operand must specify a register ‘sX’ from which the second operand will be subtracted (the value of sX will not be effected by the operation).
The second operand defines the value to be subtracted from the first operand and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the temporary 8-bit result is zero corresponding with both operand being equal or ‘matching’.
The carry flag (C) will be set if the temporary result of the subtraction is negative and hence indicating when ‘sX’ is less than the second operand.

Flag States		Comparison
Z	C	
0	0	**sX > kk** or **sX > sY**
x	1	**sX < kk** or **sX < sY**
1	x	**sX = kk** or **sX = sY**

COMPARE sX, kk temp = sX - kk

COMPARE sX, sY temp = sX - sY



Examples

```
LOAD sA, 8E
COMPARE sA, 8E
JUMP Z, equal
```

Values are equal, Z=1, C=0.

```
LOAD sA, 8E
COMPARE sA, 98
JUMP C, less_than
```

sA < 98, Z=0, C=1.

Hints

Use ‘Z’ to determine when the values are equal or ‘match’.

The ‘C’ flag can be used to determine when sX is less than the second operand. Hence, it can also be used to determine when sX is greater than or equal to the second operand’. So when comparing the contents of two registers assign them to ‘sX’ and ‘sY’ such that you can use ‘C’ to identify which is less. This will avoid the requirement to test both ‘C’ and ‘Z’ flags.

The KSPSM6 Assembler enables you to specify constants in hex, decimal and ASCII characters, e.g. COMPARE s0, “Q”

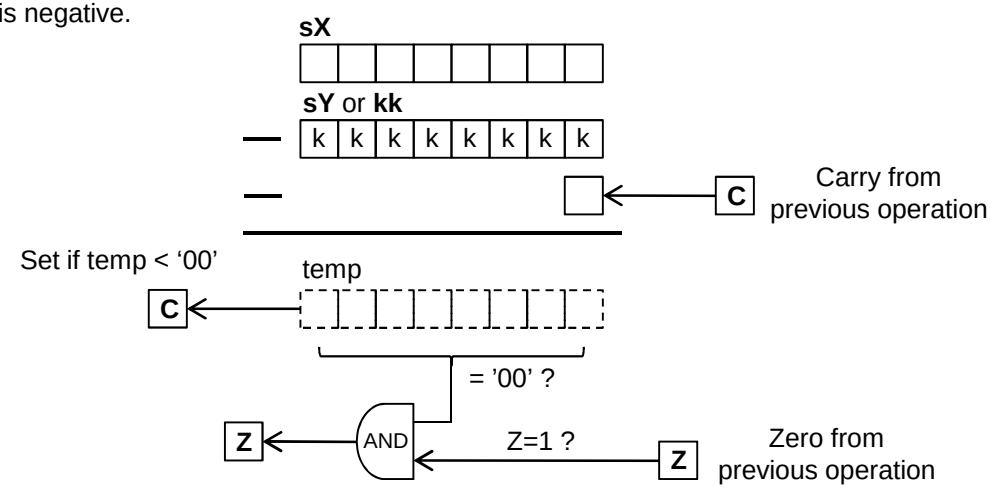
COMPARECY sX, kk
COMPARECY sX, sY

The ‘COMPARECY’ instructions are primarily intended as an extension to the basic ‘COMPARE’ instructions in order to support comparison of values more than 8-bits. The key difference from the COMPARE instructions is that the zero and carry flags are also used as inputs to the subtraction used to perform the comparison and these can influence both the 8-bit result before it is discarded and the new states of the flags. Although each register only contains an 8-bit value, any combination of registers can be used to hold larger values segmented into bytes. For example a 32-bit value can be held in a 4 registers. Although there is no restriction on which registers, and no formal way of describing the assignment, it is common practice to assign adjacent registers and refer to them as a ‘register set’ such as [sD, sC, sB, sA].

The first operand must specify a register ‘sX’ from which the second operand and carry flag will be subtracted (sX will not be effected by the operation).
The second operand defines the value to be subtracted from the first operand and can either be an 8-bit constant ‘kk’ or a register ‘sY’.
The zero flag (Z) will be set if the temporary 8-bit result is zero is zero *and* the zero flag was set prior to the COMPARECY instruction.
The carry flag (C) will be set if the temporary result of the subtraction is negative.

COMPARECY sX, kk temp = sX - kk - C

COMPARECY sX, sY temp = sX - sY - C



Examples

The meaning of the ‘C’ and ‘Z’ flags are the same following a COMPARE and COMPARECY combination of instructions used to compare multi-byte values as they are after a single 8-bit COMPARE operation.

```
LOAD sA, 7B
LOAD sB, A2
LOAD sC, 14
COMPARE sA, 7B
COMPARECY sB, A2
COMPARECY sC, 14
JUMP Z, equal
```

[sC,sB, sA] = 14 A2 7B

14A27B – 14A27B = 000000

Values are equal Z=1, C=0.

```
LOAD sA, 7B
LOAD sB, A2
COMPARE sA, 7B
COMPARECY sB, B9
JUMP C, less_than
```

[sB, sA] = A2 7B

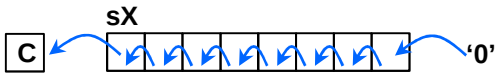
A27B – B97B = (-) E900

[sB, sA] < B97B Z=0, C=1.

SL0 sX SL1 sX SLX sX SLA sX

These instructions all shift the contents of the specified register (sX) one bit to the left. The most significant bit (MSB) is shifted out of the register into the carry flag (C). The bit that is shifted into the least significant bit (LSB) is defined by the shift left instruction that is used. The zero flag (Z) will be set only if all 8-bits of the resulting value contained in the register are zero.

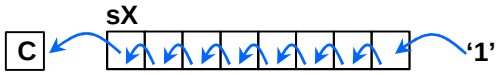
SL0 sX



Z =?

SL0 sX shifts a '0' into the LSB. The Z flag will only be set if bits(7:1) are also all '0' after the shift.

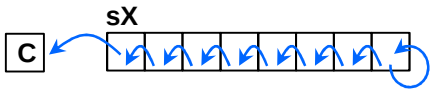
SL1 sX



Z =0

SL1 sX shifts a '1' into the LSB. This means that the Z flag will be always be cleared (Z=0) by this instruction.

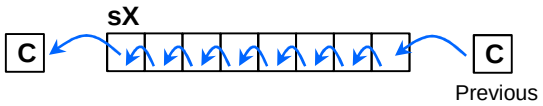
SLX sX



Z =?

SLX sX replicates the existing state of the LSB. The Z flag will only be set if all 8-bits of the register are zero.

SLA sX



Z =?

SLA sX shifts the previous state of the carry flag into the LSB at the same time that the carry flag is loaded with the MSB. The Z flag will only be set if all 8-bits of the register are zero.

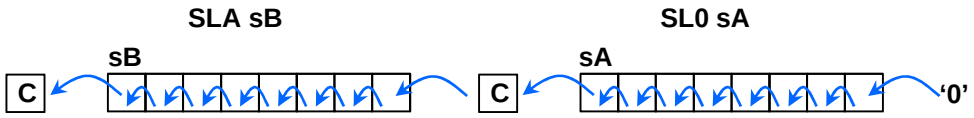
Examples

A shift left injecting a '0' has the effect of multiplying a value by 2. The 'SLA' instruction enables multi-byte values contained in multiple registers to be shifted.

```
LOAD sB, 14
LOAD sA, B5
SL0 sA
SLA sB
```

[sB,sA] = 14B5 = 5301_{10} = 0001 0100 1011 0101

[sB,sA] = 296A = 10602_{10} = 0010 1001 0110 1010




```
      LOAD sF, 00000001'b
loop: OUTPUT sF, port
      SLX sF
      JUMP NC, loop
```

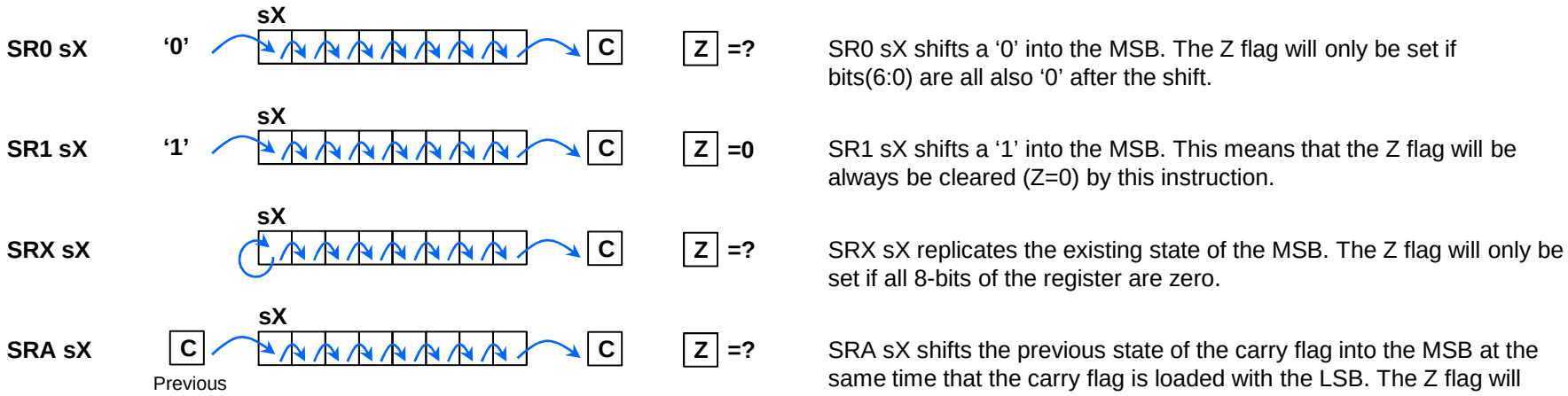
Outputs a simple pattern shown on the right hand side to the to 'port'. The process terminates when all 8-bits have been set and the final shift sets the carry flag.

```
00000001
00000011
00000111
00001111
00011111
00111111
01111111
11111111
```


XILINX

SR0 sX SR1 sX SRX sX SRA sX

These instructions all shift the contents of the specified register (sX) one bit to the right. The least significant bit (LSB) is shifted out of the register into the carry flag (C). The bit that is shifted into the most significant bit (MSB) is defined by the shift right instruction that is used. The zero flag (Z) will be set only if all 8-bits of the resulting value contained in the register are zero.



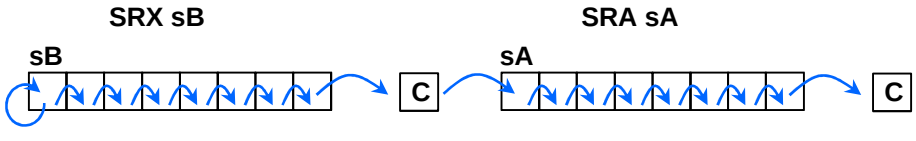
Examples

A shift right has the effect of dividing a value by 2. The 'SRA' instruction enables multi-byte values contained in multiple registers to be shifted. When 2's complement is used to represent signed values then 'SRX' implements sign extension.

```
LOAD sB, ED
LOAD sA, 2A
SL0 sA
SLA sB
```

[sB,sA] = ED2A = -4822_{10} = 1110 1101 0010 1010

[sB,sA] = F695 = -2411_{10} = 1111 0110 1001 0101




```
      LOAD sF, 10000000'b
loop: OUTPUT sF, port
      SR0 sF
      JUMP NC, loop
```

Outputs to 'port' a simple 'walking 1' pattern as illustrated on the right hand side. The process terminates when the '1' is shifted into the carry flag.

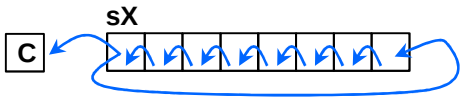
```
10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
00000000
```


XILINX

RL sX
RR sX

The ‘RL’ and ‘RR’ instructions rotate the contents of the specified register (sX) one bit to the left or right. The bit that is shifted out of on end of the register and back into the other end is also copied into the into the carry flag (C). The zero flag (Z) will be set only if all 8-bits of the register contents are zero.

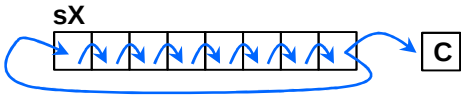
RL sX



Z =?

RL sX shifts all bits one place to the left and the MSB that is shifted out is shifted into the LSB as well as copied into the carry flag. The Z flag will only be set if all bits of the register are zero.

RR sX



Z =?

RR sX shifts all bits one place to the right and the LSB that is shifted out is shifted into the MSB as well as copied into the carry flag. The Z flag will only be set if all bits of the register are zero.

Note that because the rotate instructions only reorganize the existing contents of ‘sX’ the zero flag will only be set if ‘sX’ contained zero on entry to the rotate operation.

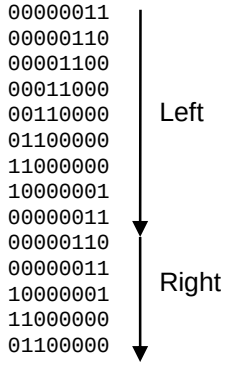
Example

Rotate operations are typically used in the generations of bit patterns or sequences such as in the control of stepper motors.

```
            LOAD s6, 03
      loop: OUTPUT s6, motor_ctrl
            CALL step_delay
            INPUT s0, direction
            TEST s0, 01
            JUMP NZ, move_right
            RL s6
            JUMP loop
move_right: RR s6
            JUMP loop
```

In this example we can imagine a stepper motor that has 8 coils arranged in a circle such that the coil mapped to bit0 is adjacent to the coil mapped to bit7. The position of the motor is defined by the coils being energised and in this case it is beneficial to energise two adjacent coils of a motor at the same time (hence the initial value of 03 loaded into ‘s6’).

An input is sampled to determine in which direction the motor should rotate and this is translated into the direction in which the “11” pattern is rotated either to the left or right.



REGBANK A
REGBANK B

KCPSM6 actually has 32 registers that are arranged into 2 banks of 16 registers called bank ‘A’ and bank ‘B’. Only one bank can be active at a given time and all instructions (except ‘STAR’) can only perform operations involving the registers is the active bank. To put it another way the registers in the inactive bank are almost completely isolated and their contents are unaffected by instructions modifying values within the registers of the active bank.

Following device configuration or an active High reset pulse on the ‘reset’ input of the KCPSM6 macro bank ‘A’ will be the active bank. Hence KCPSM6 can initially be considered to have 16 registers and this will be adequate for many applications.

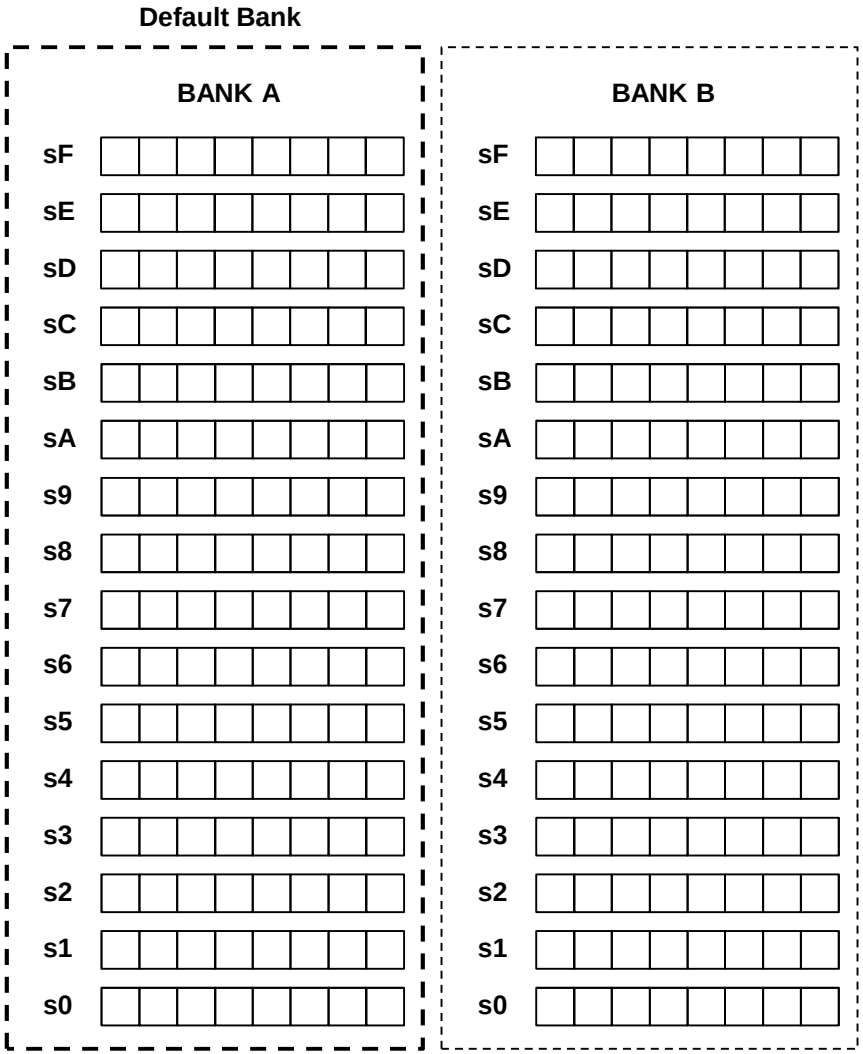
The REGBANK instruction can be used to select which bank is to be active and therefore assign the other bank to being inactive. There is only one carry flag and one zero flag neither of which is effected by the bank selection.

REGBANK A - Select bank A active (or restore default bank).

REGBANK B - Select bank B active.

Hint – When selecting a different bank there is no effect on the contents of any registers in either bank. However, it will almost certainly appear as if the contents of all registers change as the swap is made.

Hint - If you use the NAMEREG directive in your code then you will probably want to assign different names to the registers following the REGBANK instruction to reflect that you are no longer accessing the same information.



STAR sX, sY

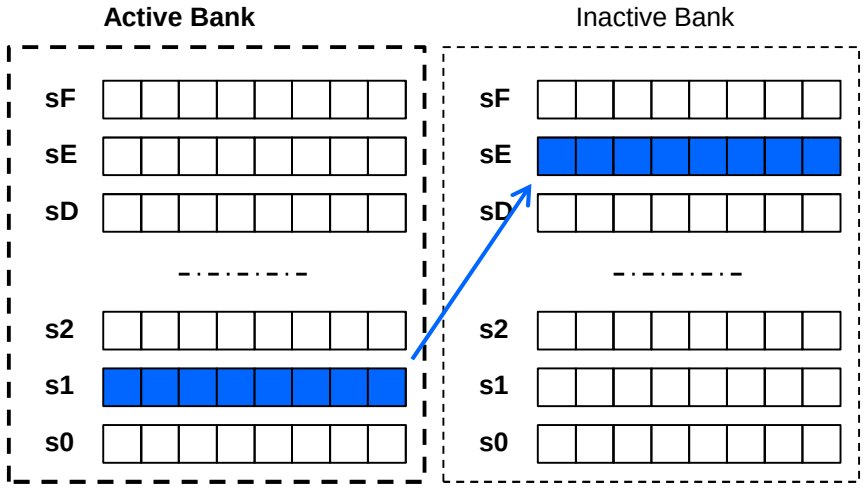
Apart from bank ‘A’ being the default on power up or following a reset you are completely free to select back ‘A’ or bank ‘B’ as an when you wish using the REGBANK instruction. All instructions only operate on the registers in the actively selected bank which preserves the values in the inactive bank. This is typically of value when creating a subroutine that implements an intense task where the use of many registers to manipulate data makes the task easier. It is also very appealing when servicing an interrupt as it can really help to ensure that the contents of registers being used anywhere else in the program at the time of the interrupt are not disturbed (this is covered in more detail in the interrupt section of this guide).

In KCPSM3 that only has one bank of registers, a common technique is to preserve the contents of registers in scratch pad memory before those registers are used again by a subroutine. The values are then fetched from memory to restore those values before returning to the main program. This is still a perfectly valid technique in KCPSM6 programs but it can result in a significant number of STORE and FETCH instructions consuming code space and slowing program execution. By switching to the ‘B’ bank of registers at the start of an intense subroutine or when servicing and interrupt you could effectively provide yourself with 16 temporary registers in one instruction cycle (2 system clock cycles) automatically preserving the contents of registers in bank ‘A’.

Although it is useful to have two banks of registers that are isolated and independent this also presents a challenge when it comes to data being passed between a main program and a subroutine. Once again a solution is to assign particular scratch pad memory locations which are then accessed by both sections of code using different register banks but this can be somewhat constraining as well. For this reason the ‘Send To Alternate Reregister’ or ‘STAR’ instruction provides you with a way to pass information from a register in the active bank (sY) to a register in the inactive bank (sX).

STAR sX, sY

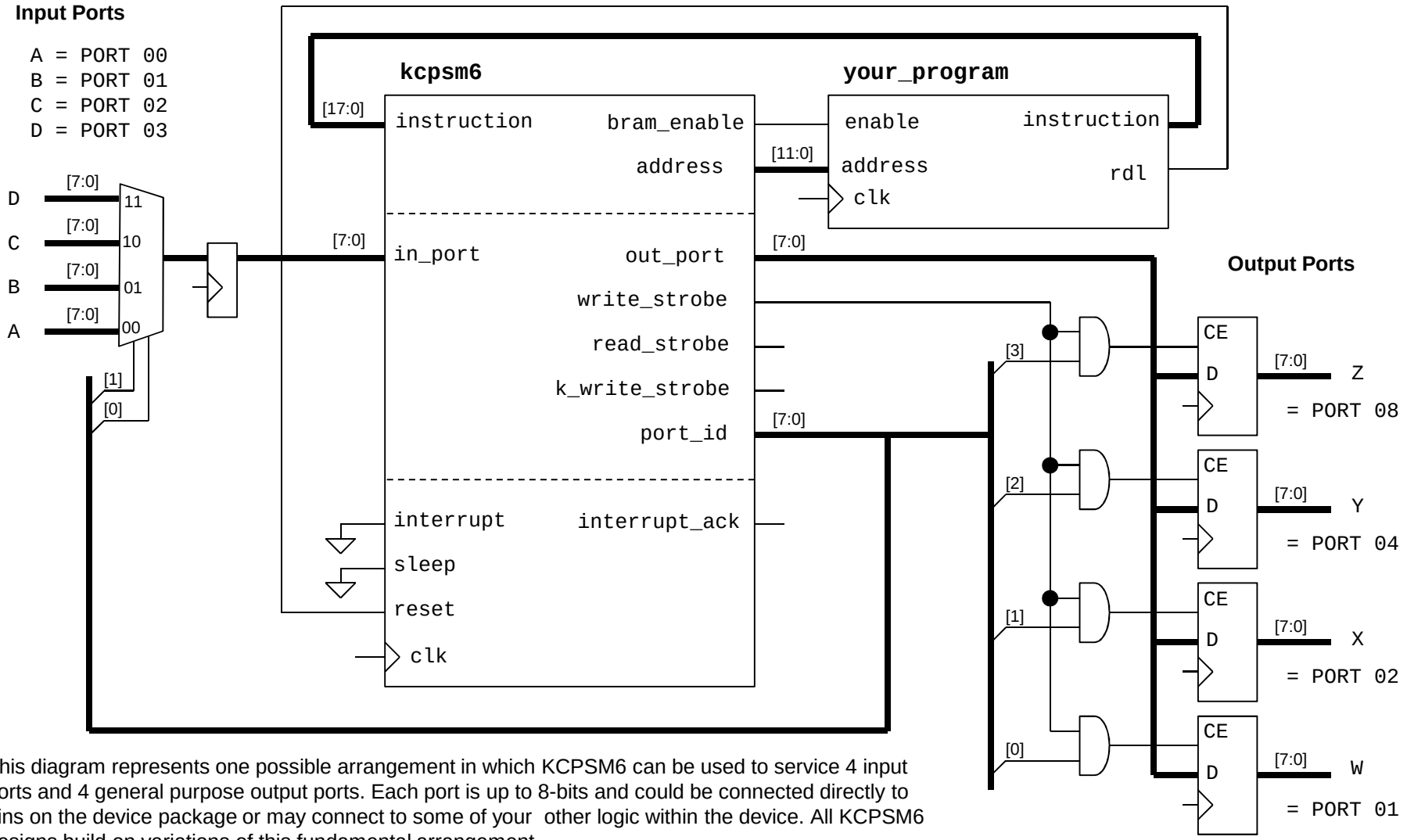
Hint – ‘STAR sX, sY’ is almost exactly the same as a ‘LOAD sX, sY’ and also has no effect on the states of the flags. However it should be recognised that ‘STAR s0, s0’ is definitely not the equivalent of a no-operation because each reference to ‘s0’ is in a different bank and therefore the contents of ‘s0’ in the inactive bank will probably be changed.



Example `STAR sE, s1` ‘sE’ in the inactive back is loaded with a copy of ‘s1’ in the active bank (illustrated in the diagram above).

ASSEMBLER CODING REQUIREMENT The alternate register ‘sX’ *must be* specified using a default name ‘s0’ to ‘sF’. Any NAMEREG directives do not apply to the specification of ‘sX’. The current active register ‘sY’ *must be* specified using an active name for a register (i.e. NAMEREG does apply as normal). These are deliberate coding rules intended to minimise the probability of coding mistakes (i.e. They force you to think carefully about what bank is active).

General Purpose I/O Ports

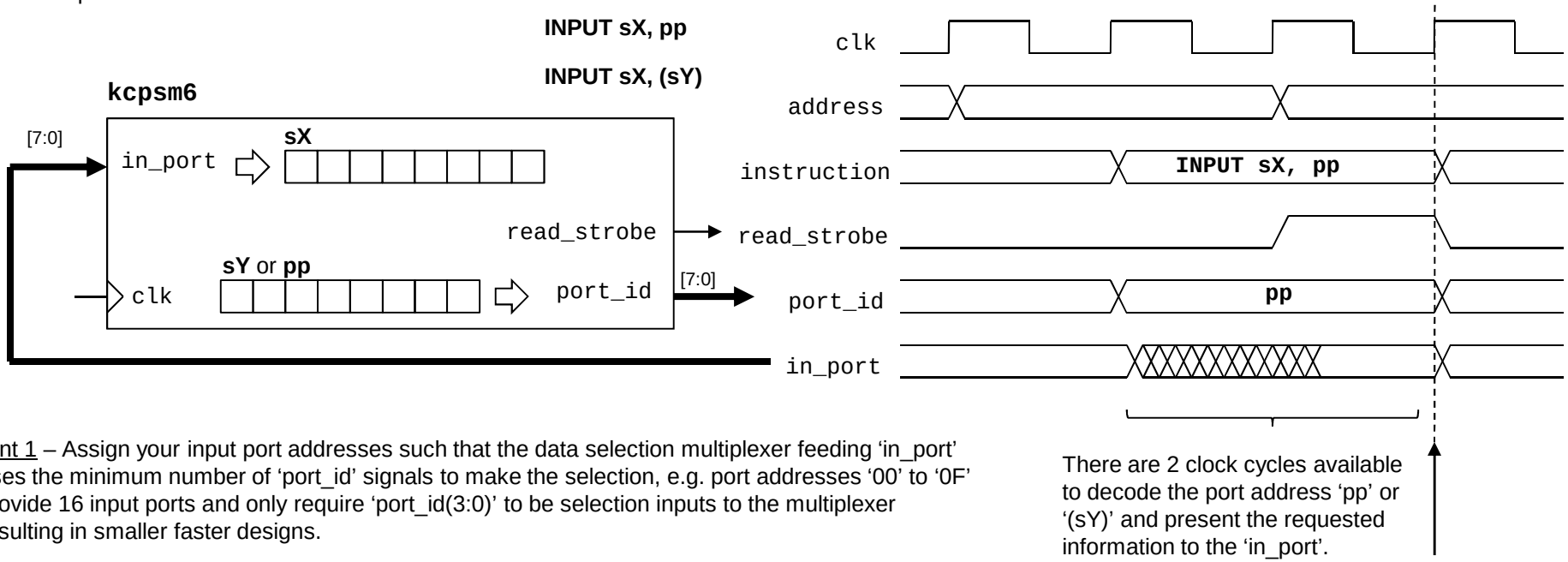


This diagram represents one possible arrangement in which KCPSM6 can be used to service 4 input ports and 4 general purpose output ports. Each port is up to 8-bits and could be connected directly to pins on the device package or may connect to some of your other logic within the device. All KCPSM6 designs build on variations of this fundamental arrangement.

XILINX

INPUT sX, pp
INPUT sX, (sY)

An ‘INPUT’ instruction enables KCPSM6 to read information from the from your hardware design into a register ‘sX’ using a general purpose input port specified by an 8-bit constant value ‘pp’ or the contents of another register ‘(sY)’. KCPSM6 presents the port address defined by ‘pp’ or ‘(sY)’ on ‘port_id’ and your hardware interface is then responsible for selecting and presenting the appropriate information to the ‘in_port’ so that it can be captured into the ‘sX’ register. An active High (‘1’) synchronous pulse is also generated on the ‘read_strobe’ pin and may be used by the hardware interface to confirm when a particular port has been read.



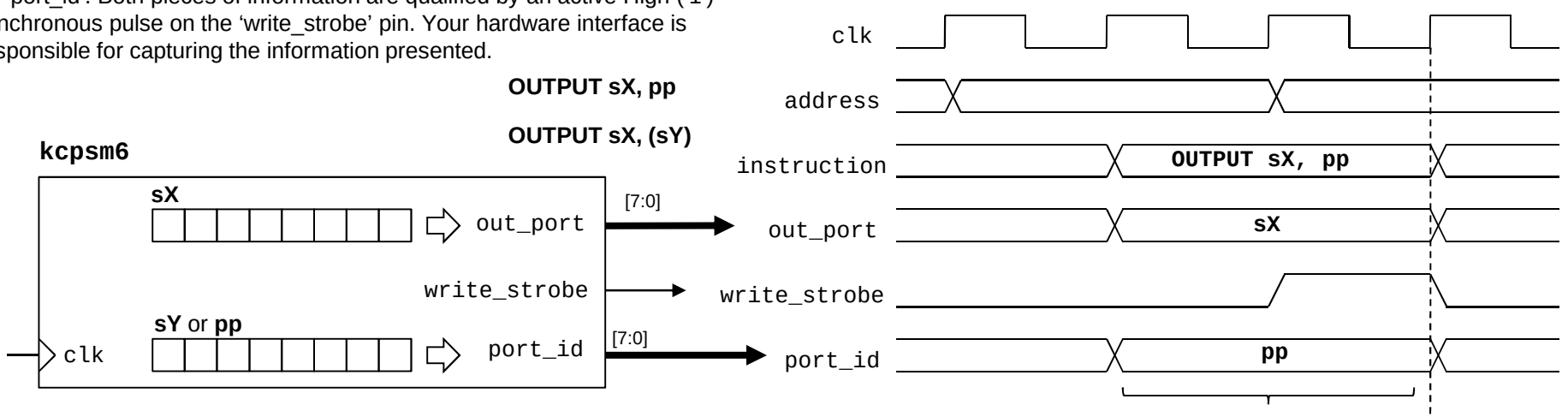
Hint 1 – Assign your input port addresses such that the data selection multiplexer feeding ‘in_port’ uses the minimum number of ‘port_id’ signals to make the selection, e.g. port addresses ‘00’ to ‘0F’ provide 16 input ports and only require ‘port_id(3:0)’ to be selection inputs to the multiplexer resulting in smaller faster designs.

Hint 2 – Unless there is a specific reason not to, the input data selection multiplexer should include a pipeline register (i.e. your case statement should be within a clocked process). In this way the data is selected during the first clock cycle of ‘port_id’ and presented to ‘in_port’ during the second clock cycle. Failure to define a pipeline register anywhere in the ‘port_id’ to ‘in_port’ path is the most common reason for PicoBlaze designs failing to meet the required performance (a ‘false path’ for one clock cycle) .

Hint 3 – ‘read_strobe’ can be ignored in most cases and never needs to be part of the multiplexer feeding ‘in_port’. However, some functions such as a FIFO buffer do need to know when they have been read and it is in those situations that ‘read_strobe’ together with a decode of the appropriate value of ‘port_id’ would be used to generate a “port has been read” pulse to confirm when a read has taken place.

OUTPUT sX, pp
OUTPUT sX, (sY)

An ‘OUTPUT’ instruction is used to transfer information from a register ‘sX’ to a general purpose output port specified by an 8-bit constant value ‘pp’ or the contents of another register ‘(sY)’. KCPSM6 presents the contents of the register ‘sX’ on ‘out_port’ and the port address defined by ‘pp’ or ‘(sY)’ is presented on ‘port_id’. Both pieces of information are qualified by an active High (‘1’) synchronous pulse on the ‘write_strobe’ pin. Your hardware interface is responsible for capturing the information presented.



Note that ‘out_port’ and ‘port_id’ will vary during the execution of other instructions but ‘write_strobe’ will only be active during an OUTPUT instruction.

Hint – In most cases a fixed port address ‘pp’ is used so CONSTANT directives provide an ideal why track your port assignments and make your code easier to write, understand and maintain.

Examples

```
CONSTANT LED_port, 05
;
LOAD s3, 3A
OUTPUT s3, LED_port
```

If you want to keep your designs small and fast then assign port addresses that facilitate smaller logic functions.

In this example a set of 8 LEDs are mapped to port 05 hex and only 3-bits of ‘port_id’ together with ‘write_strobe’ are decoded.

VHDL

```
if clk'event and clk = '1' then
  if write_strobe = '1' then
    if port_id(2 downto 0) = “101” then
      led <= out_port;
    end if;
  end if;
end if;
```

```
OUTPUT s6, (s2)
OUTPUT s4, 40
OUTPUT sB, 64’d
```

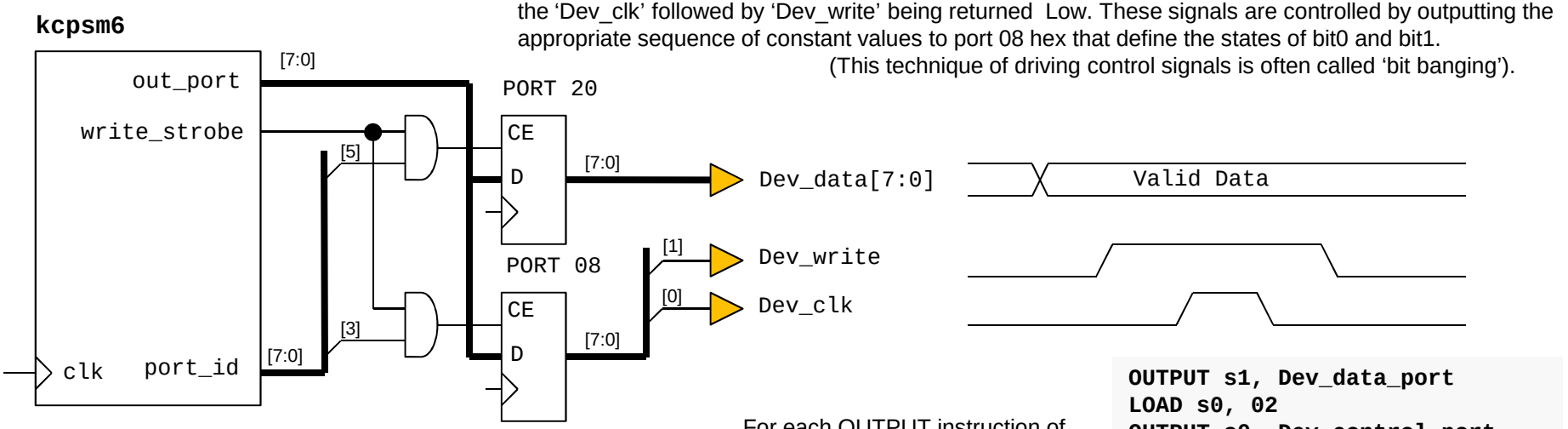
Decimal values can be used to specify port addresses but hex or binary values are normally easier to work with when defining the hardware.

Constant-Optimised Output Ports

In order to understand the motive for the constant-optimised ports and to know when it is better to use them, it is necessary to appreciate the situations in which the general output ports can adversely effect the size of your program code and/or result in lower performance. The 'OUTPUT sX, pp' and 'OUTPUT sX, (sY)' instructions associated with the general purpose output ports both require that the value to be written to the port to be held in a register 'sX'. This is ideal when the value is a variable in your system but when you want to send a constant value, or more likely, a series of constant values to a port the act of loading 'sX' each time increases code size and reduces performance. In many applications this overhead can be tolerated and you should feel no pressure to adapt your design and code to use the constant-optimised ports unless you really want to. However, using constant-optimised ports appropriately can make code easier to write and avoid the code size and performance overhead associated with general purpose output ports when necessary.

Using General Purpose Output ports……

In this example KCPSM6 is required to write 8-bit data to an external device. The data is naturally variable and is presented to the device interface by outputting to port 20 hex. Then KCPSM6 is required to generate the correct sequence of control signals; 'Dev_write' is set High before a pulse is generated on the 'Dev_clk' followed by 'Dev_write' being returned Low. These signals are controlled by outputting the appropriate sequence of constant values to port 08 hex that define the states of bit0 and bit1.
(This technique of driving control signals is often called 'bit banging').



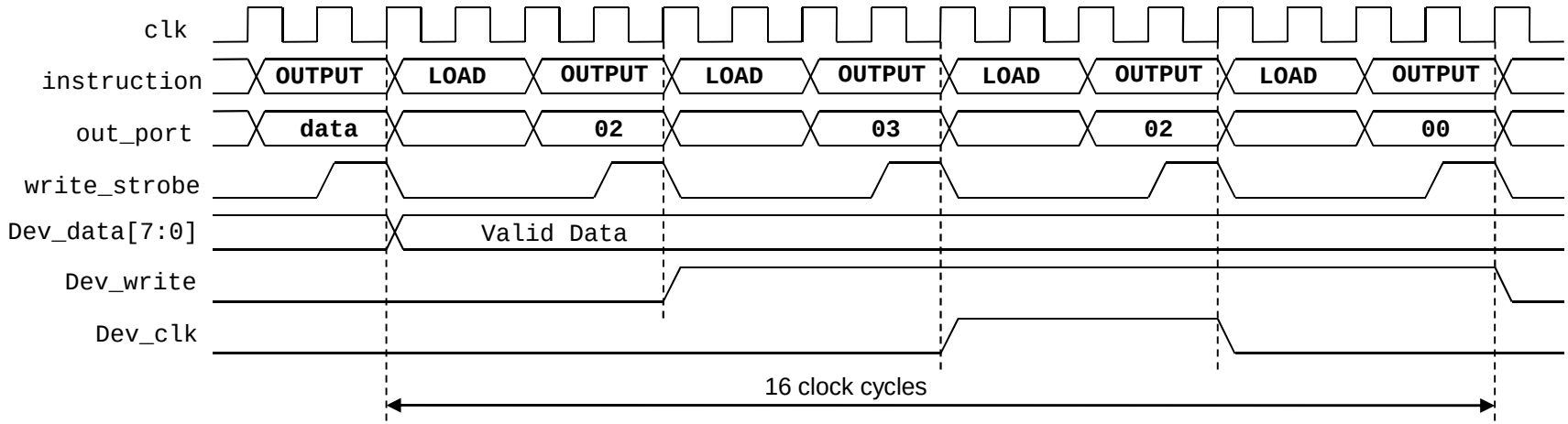

```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

For each OUTPUT instruction of the control sequence waveform there is a corresponding LOAD instruction that prepares 's0' with the required constant value.

```
OUTPUT s1, Dev_data_port
LOAD s0, 02
OUTPUT s0, Dev_control_port
LOAD s0, 03
OUTPUT s0, Dev_control_port
LOAD s0, 02
OUTPUT s0, Dev_control_port
LOAD s0, 00
OUTPUT s0, Dev_control_port
```

Constant-Optimised Output Ports

The timing diagram for the code using the general purpose output ports shows that it takes 16 system clock cycles to generate the control sequence because every instruction takes 2 clock cycles and every OUTPUT instruction requires a corresponding LOAD instruction to initialise 'sX' ('s0' was used in the example). It can also be seen that this results in 4 clock cycles between each transition of the control sequence.



There are a number of applications where it is beneficial that KPCSM6 slows down the generation of waveforms. For example, the communication rate with an SPI Flash memory device may be 33MHz maximum. So if your system clock was 200MHz you would be looking to divide that by at least a factor or 6 and KCPSM6 could help to achieve that naturally. However, if you require higher 'bit banging' performance without just increasing the system clock frequency then clearly there is a limit when using the general purpose output ports.

```
OUTPUT s1, Dev_data_port
LOAD s2, 02
LOAD s3, 03
LOAD s0, 00
OUTPUT s2, Dev_control_port
OUTPUT s3, Dev_control_port
OUTPUT s2, Dev_control_port
OUTPUT s0, Dev_control_port
```

One potential workaround that has been used in KCPSM3 based designs in the past, and is still applicable to KCPSM6 designs, is to reorder your code. As shown on the left, the constant values have been pre-loaded into a set of registers so that the waveform can be generated with a burst of sequential OUTPUT instructions. Whilst this does result in the highest possible 'bit banging' transition rate of the signals during the actual generation of the sequence it also requires more registers to be used and the same amount of time is required to execute the code overall.

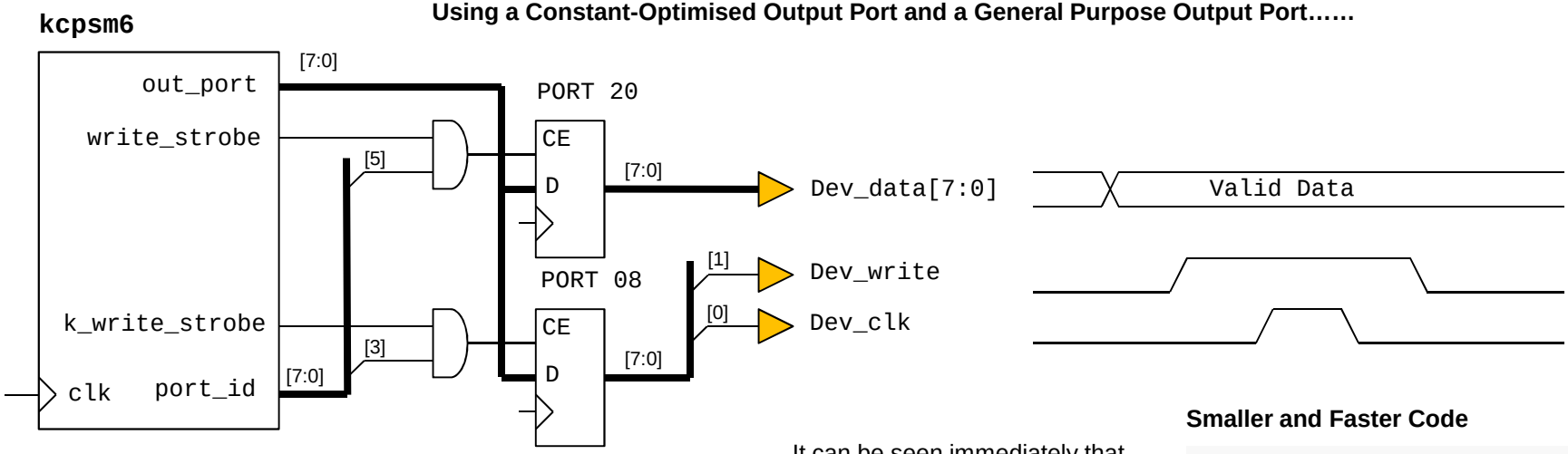
Hint – To generate single clock cycle pulses you can use the single clock cycle 'write_strobe' qualified by the 'port_id' rather than set and reset a data bit of a full output port.

Constant-Optimised Output Ports

KCPSM6 provides up to 16 constant-optimised output ports. From a hardware perspective these are used in an identical way to the general purpose output ports except that 'k_write_strobe' is used to qualify the port address which is presented on port_id[3:0]. Hence only port addresses '0' to 'F' (0'd to 15'd) can be used and port_id[7:4] should be ignored. Good optimum designs will allocate output port addresses to minimise the decoding of 'port_id' so this should not pose any challenges.

Returning to the same example of writing data to an external device we can see that port 08 hex has now been allocated to a constant-optimised output port by using the 'k_write_strobe' whilst port 20 hex is still associated with 'write_strobe' because the data is naturally variable. So there is very little difference in the hardware as long as you remember that only port_id[3:0] are defined during an OUTPUTK instruction. Note also that you could now have two different output ports with the same address; on for variable data and the other for constant values (see page 79).

Using a Constant-Optimised Output Port and a General Purpose Output Port……



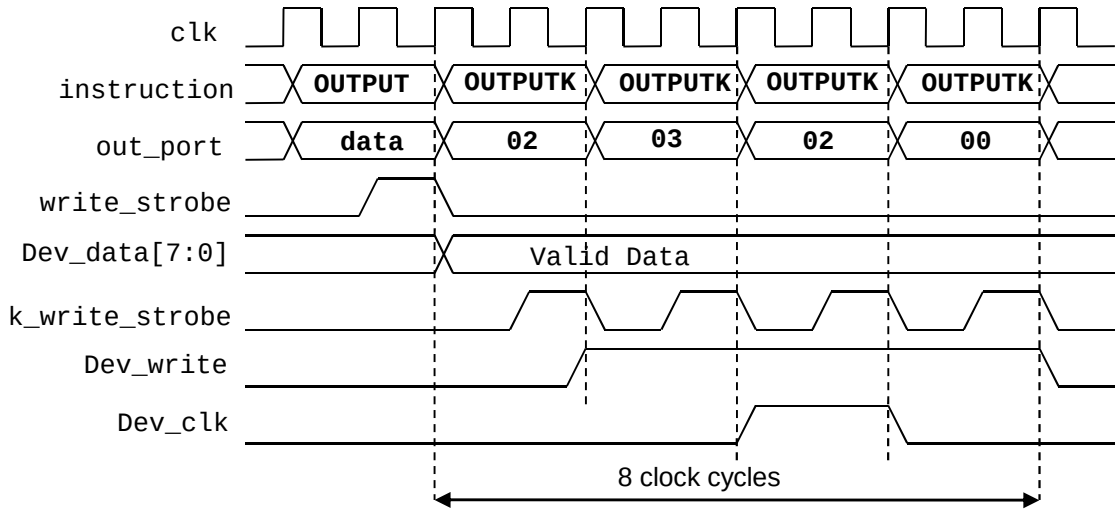

```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

It can be seen immediately that all the LOAD instructions have been eliminated saving code space and reducing the execution time. This also means that register 's0' used previously to define the sequence of values is now free for another purpose.

Smaller and Faster Code

```
OUTPUT s1, Dev_data_port
OUTPUTK 02, Dev_control_port
OUTPUTK 03, Dev_control_port
OUTPUTK 02, Dev_control_port
OUTPUTK 00, Dev_control_port
```

Constant-Optimised Output Ports
OUTPUT kk, p



Smaller and Faster Code

```
OUTPUT s1, Dev_data_port
OUTPUTK 02, Dev_control_port
OUTPUTK 03, Dev_control_port
OUTPUTK 02, Dev_control_port
OUTPUTK 00, Dev_control_port
```

This timing diagram clearly shows the performance advantage when using a constant-optimised output port for a 'bit banging' application. The example control sequence is now completed in 8 rather than 16 clock cycles. More significantly, the standard transition rate is every instruction or 2 system clock cycles. All without the need to use any registers.

OUTPUTK kk, p

The OUTPUTK instruction has two operands. The first operand is the 8-bit constant value 'kk' that will be presented on 'out-port' and therefore must be in the range '00' to 'FF' hex. The second operand must specify the port address that will be presented on port_id[3:0] and therefore must be in the range '0' to 'F' hex.

Examples

```
CONSTANT token, 61
CONSTANT control_port, 0A

OUTPUTK 61, A
OUTPUTK 97'd, 10'd
OUTPUTK "a", A
OUTPUTK token, control_port
```

These examples show how the KCPSM6 assembler enables the constant and port to be defined and specified in multiple ways. All four 'OUTPUTK' instructions shown are actually the same!

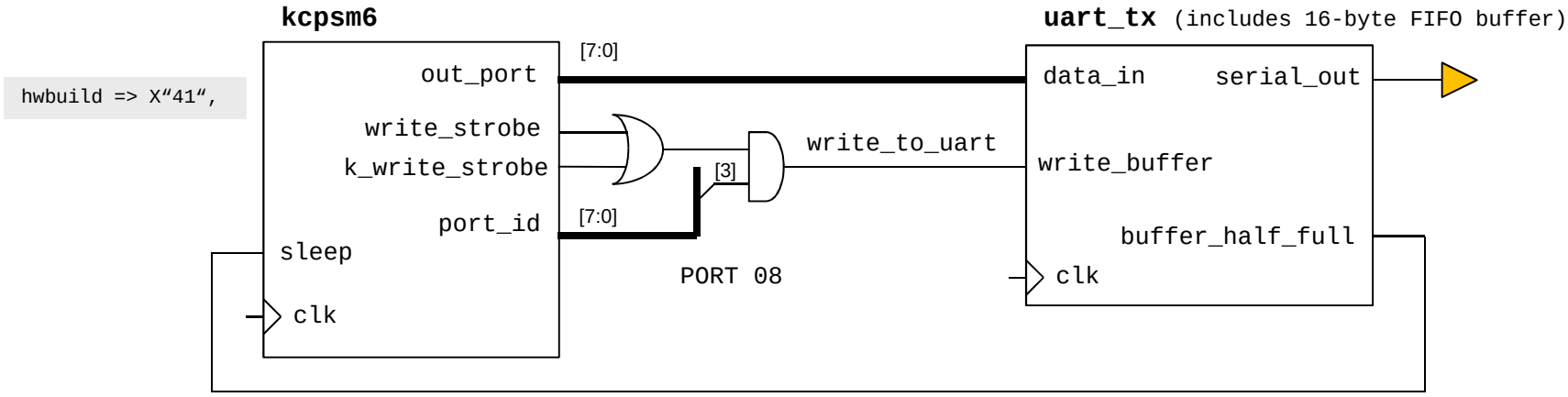
The constant value 'kk' can be specified immediately using hex, decimal or an ASCII character. Alternatively the name allocated to a constant by a CONSTANT directive can be used.

The port address 'p' can also be specified immediately using hex or decimal but remember that this can only be in the range '0' to 'F' (0'd to 15'd). Likewise, the name of constant defined by a CONSTANT directive can be used providing that the value assigned to it also falls within the required range.

Implementing Hybrid Output Ports

Whenever variable data needs to be sent to an output port then a general purpose output port must be used. However, mainly to improve code density the 'OUTPUTK kk, p' instruction is more suitable in some situations so it then becomes desirable to deliver constants to the same port. This can be achieved simply by allocating the same port address (in the range '00' to '0F' hex) to be used by both 'OUTPUT sX, pp' and 'OUTPUTK kk, p' instructions and implementing a hybrid port in hardware.

An example of a hybrid port is shown below. In this case KCPSM6 is required to send information to a UART transmitter to be observed on a PC terminal. Not surprisingly, the information will be a series of ASCII characters but many of these will be pre-defined strings or constants whilst others will represent the variable data to be displayed. This example also illustrates a possible use of the 'sleep' control and the 'STRING' directive in the KCPSM6 assembler.



Hybrid port decode in VHDL

```
write_to_uart <= (k_write_strobe or write_strobe) and port_id(3);
```

In order to create a hybrid port the port address must be in the range '00' to '0F' hex and in this example 08 hex has been used in a optimised decode of 'port_id' (i.e. only port_id[3] is actually being observed to minimise the logic function performing the decode). Then both 'write_strobe' and 'k_write_strobe' are used to qualify the port address so that either an 'OUTPUT sX, 08' or an 'OUTPUT kk, 8' instruction will result in data being written to the FIFO buffer within the UART transmitter macro.

The circuit diagram also shows how the 'half full' status output from the FIFO buffer could be used to make KCPSM6 wait (sleep) each time the buffer starts to fill up and this hardware form of handshaking is important if code density is to be achieved as we will see on the next page.…

Implementing Hybrid Output Ports & Text Strings

PSM file

```
CONSTANT UART_Tx_port, 08
;
STRING hw_intro$, "Hardware Build: “
OUTPUTK hw_intro$, UART_Tx_port
HWBUILD s0
OUTPUT s0, UART_Tx_port
OUTPUTK 0D, UART_Tx_port
```

LOG file

```
000  2B488    OUTPUTK 48[hw_intro$:"H"], 8[UART_Tx_port]
001  2B618    OUTPUTK 61[hw_intro$:"a"], 8[UART_Tx_port]
002  2B728    OUTPUTK 72[hw_intro$:"r"], 8[UART_Tx_port]
003  2B648    OUTPUTK 64[hw_intro$:"d"], 8[UART_Tx_port]
004  2B778    OUTPUTK 77[hw_intro$:"w"], 8[UART_Tx_port]
005  2B618    OUTPUTK 61[hw_intro$:"a"], 8[UART_Tx_port]
006  2B728    OUTPUTK 72[hw_intro$:"r"], 8[UART_Tx_port]
007  2B658    OUTPUTK 65[hw_intro$:"e"], 8[UART_Tx_port]
008  2B208    OUTPUTK 20[hw_intro$:" "], 8[UART_Tx_port]
009  2B428    OUTPUTK 42[hw_intro$:"B"], 8[UART_Tx_port]
00A  2B758    OUTPUTK 75[hw_intro$:"u"], 8[UART_Tx_port]
00B  2B698    OUTPUTK 69[hw_intro$:"i"], 8[UART_Tx_port]
00C  2B6C8    OUTPUTK 6C[hw_intro$:"l"], 8[UART_Tx_port]
00D  2B648    OUTPUTK 64[hw_intro$:"d"], 8[UART_Tx_port]
00E  2B3A8    OUTPUTK 3A[hw_intro$:":"], 8[UART_Tx_port]
00F  2B208    OUTPUTK 20[hw_intro$:" "], 8[UART_Tx_port]
010  14080    HWBUILD s0
011  2D008    OUTPUT s0, 08[UART_Tx_port]
012  2B0D8    OUTPUTK 0D, 8[UART_Tx_port]
```

This code uses the hybrid port to display the hardware build state on the PC terminal display.

```
Hardware Build: A
```

The code exploits the STRING directive to describe the sequence of 16 constant values required to send ‘Hardware Build: ’ to the UART using ‘OUTPUTK’ instructions. It then loads ‘s0’ with the ‘hwbuild’ value defined as 41 hex (character “A”) using the KCPSM6 generic (see pages 34 and 100) which it sends to the UART using an ‘OUTPUT’ instruction. The communication is completed by sending a carriage return (0D hex) using an ‘OUTPUTK’ instruction.

The LOG file shows how the text string has been used to expand the code into multiple ‘OUTPUTK’ instructions. Under normal operating conditions the complete 19 instruction sequence will execute in just 38 system clock cycles. It is therefore vital that whatever you are sending data to has the capability of receiving information at that speed. In this example the UART transmitter only has a 16 character FIFO buffer so hardware handshaking exploiting the ‘sleep’ control is the solution. An alternative would be to have a larger FIFO buffer and to be sure it had adequate free space before starting to send the characters (i.e. test for FIFO empty state prior to sending the burst of information).

Note that if you were to implement a software based check of the FIFO status by reading an input port then you would increase the code to the same size as it would be if you only used a general purpose output port . Therefore hardware handshaking schemes using ‘sleep’ or interrupts are the key to slowing down ‘OUTPUTK’ sequences if code density is the key objective.

```
CALL test_FIFO_full
OUTPUTK "H", 8
CALL test_FIFO_full
OUTPUTK "a", 8
```

```
LOAD s1, "H"
CALL send_to_UART
LOAD s1, "a"
CALL send_to_UART
```

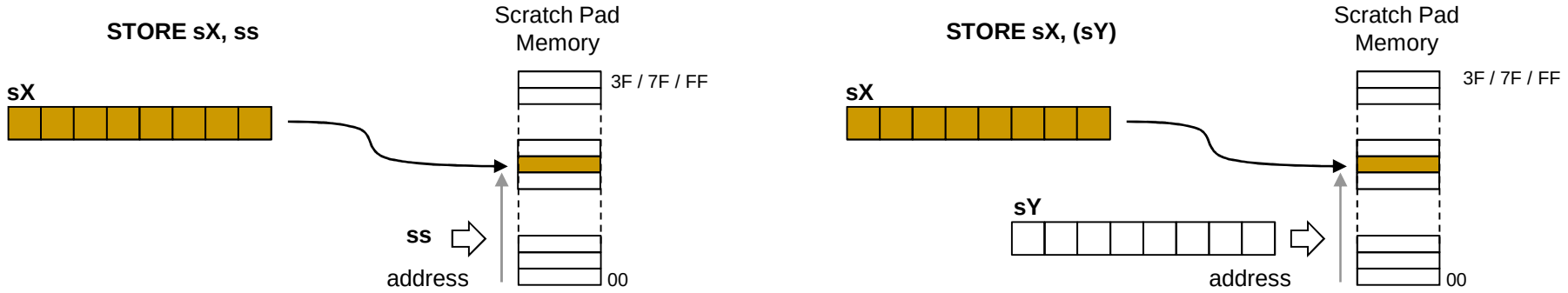
Hint – Look also at string support using ‘LOAD&RETURN’ (page 99).

Hint – TABLE Directive described in ‘all_kcpsm6_syntax.psm’.

STORE sX, ss
STORE sX, (sY)

The store instructions write the contents of a register ‘sX’ into the scratch pad memory (SPM). The location (or address) within the SPM into which the register contents are written can be defined by a specific address ‘ss’ or by the contents of a second register ‘(sY)’. The contents of the register and the states of the zero and carry flags are not effected by the operation.

The default setting of KCPSM6 provides 64-bytes of scratch pad memory with locations 00 to 3F hex (0’d to 63’d). If more memory is required then the ‘scratch_pad_memory_size’ generic can be set to ‘128’ or ‘256’ to increase the size to 128-bytes (locations 00 to 7F) or 256 bytes (locations 00 to FF). It is your responsibility to ensure that you only write to locations that physically exist.



Examples

```
CONSTANT status, 12’d
CONSTANT buffer_start, 30
```

Hint – The CONSTANT directive can be used to assign meaningful names to specific locations. The constants defined in this example are then used in the code examples below

```
INPUT s0, system_state
AND s0, 1F
STORE s0, status
```

The status of a system is read from a port and after the 5 least significant bits have been isolated the value is stored in a SPM location 0C (12’d).

```
        LOAD s1, buffer_start
read8:  INPUT s0, data_in
        STORE s0, (s1)
        ADD s1, 01
        COMPARE s1, 38
        JUMP NZ, read8
```

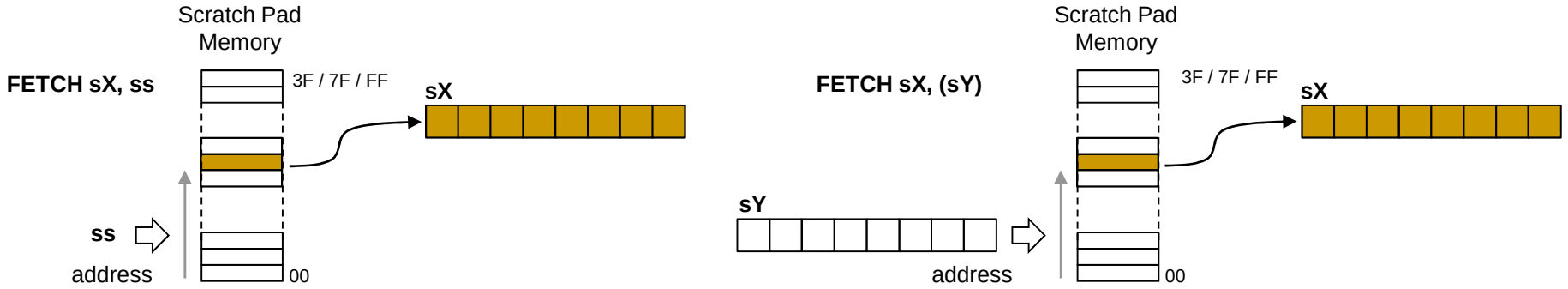
With ‘s1’ acting as a memory pointer and counter, this code reads 8 bytes of data from a port and stored it in a buffer formed of SPM locations 30 to 37 hex.

Fact – If you do write to a location larger than the size of the scratch pad memory available then the location specified in the STORE instruction will alias down into the active range and write the information at that location. For example, if ‘STORE s3, 5A ‘ is executed when the size of memory is 64 bytes then the contents of ‘s3’ will be written to location 2A hex (5A = 0101 1010 but only the lower 6-bits are used to address up to 64 locations and 10 1010 = 2A).

FETCH sX, ss
FETCH sX, (sY)

The fetch instructions read the contents of a location of scratch pad memory (SPM) into a register ‘sX’ into The SPM location (or address) to be read can be defined by a specific address ‘ss’ or by the contents of a second register ‘(sY)’. The contents of the SPM and the states of the zero and carry flags are not effected by the operation.

The default setting of KCPSM6 provides 64-bytes of scratch pad memory with locations 00 to 3F hex (0’d to 63’d). If more memory is required then the ‘scratch_pad_memory_size’ generic can be set to ‘128’ or ‘256’ to increase the size to 128-bytes (locations 00 to 7F) or 256 bytes (locations 00 to FF). It is your responsibility to ensure that you only read locations that physically exist.



Examples

```
CONSTANT status, 12’d
CONSTANT buffer_start, 30
```

Hint – The CONSTANT directive can be used to assign meaningful names to specific locations. The constants defined in this example are then used in the code examples below

```
FETCH s0, status
AND s0, 02
SR0 s0
OUTPUT s0, ok_port
```

Bit1 of the status stored in SPM location 0C (12’d) is isolated and used to set bit0 of output ‘ok_port’. Note how the full status information remains unchanged in the SPM.

```
        LOAD s2, 00
        LOAD s1, buffer_start
chksum8: FETCH s0, (s1)
        ADD s2, s0
        ADD s1, 01
        COMPARE s1, 38
        JUMP NZ, chksum8
```

With ‘s1’ acting as a memory pointer and counter, this code reads the 8 bytes of data stored in SPM locations 30 to 37 hex and adds them together (ignoring any overflow) to form a checksum value in ‘s2’.

Fact – If you do read from a location larger than the size of the scratch pad memory available then the location specified in the FETCH instruction will alias down into the active range and read information from that location. For example, if ‘FETCH s3, 5A ‘ is executed when the size of memory is 64 bytes then ‘s3’ will be loaded with the contents of SPM location 2A hex (5A = 0101 1010 but only the lower 6-bits are used to address up to 64 locations and 10 1010 = 2A).

ENABLE INTERRUPT
DISABLE INTERRUPT

See also pages 40-44

These instructions are used to control when interrupts are allowed to happen. Following device configuration or the application of a reset to the KCPSM6 macro the program starts executing from address zero and interrupts are disabled. Quite simply, this means that a High level on the 'interrupt' input will be ignored. The 'ENABLE INTERRUPT' instruction is used to enable interrupts by setting the interrupt enable flag (IE = 1). Hence this instruction need to be included at a suitable point in your code to activate the 'interrupt' input such that KCPSM6 will react to an interrupt request. 'ENABLE INTERRUPT' has no other effects.

INTERRUPT ENABLE IE ← '1'

Important – You should never execute an 'ENABLE INTERRUPT' within your ISR (i.e. anywhere between the interrupt vector and the RETURNI instruction). Once one interrupt can be serviced at a time and if you re-enable interrupts before the end of the ISR then there is every risk that another interrupt may occur.

The 'DISABLE INTERRUPT' instruction is used to disable interrupts interrupts by clearing the interrupt enable flag (IE = 0). This would typically be used to temporarily avoid the execution of critical section of code from being interrupted. 'DISABLE INTERRUPT' has no other effects.

DISABLE INTERRUPT IE ← '0'

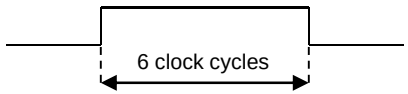
Hint – It is considered good coding practice if these instructions are only executed when actually modify the state of the interrupt enable flag. Whilst it does not cause a problem to execute the instruction in a way that confirms the state (e.g. using 'ENABLE INTERRUPT' when IE is already '1') such a coding style makes it less clear at what points you in your code interrupts are enabled and disabled and this can lead to confusion when debugging in the long term.

Examples

```
          TEST s6, 02
          JUMP NZ, no_pulse
          DISABLE INTERRUPT
          OUTPUTK 01, trigger_port
          LOAD s0, s0
          LOAD s0, s0
          OUTPUTK 00, trigger_port
          ENABLE INTERRUPT
no_pulse: LOAD s3,  JUMP Z,
```

This section of code is taken from a program at a point when interrupts are enabled and therefore subject to interruption at any time that the interrupt input is driven High.

The state of Bit1 of register 's6' is tested and if it is High then a pulse is generated on Bit0 of 'trigger_port'. A pair of 'LOAD s0, s0' instructions are used to stretch the pulse to be exactly 6 clock cycles in duration (3 instructions).



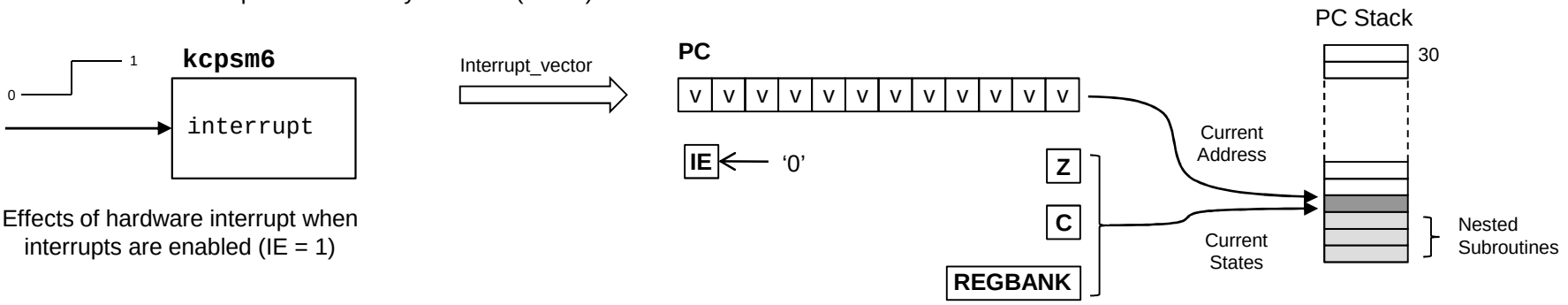
If an interrupt were to occur whilst generating the pulse then its duration could be increased considerably and in this case that was unacceptable. So to ensure that the pulse would always be 6 clock cycles interrupts are temporarily disabled only whilst the time critical is executed.

Another example would be to temporarily disable interrupts whist the main program reads information from scratch pad memory where that information is updated by the ISR. This would ensure that the information read is a complete set and not a mixture of the information resulting from 2 separate interrupts.

RETURNI ENABLE
RETURNI DISABLE

See also pages 40-44

When an interrupt occurs the program counter is loaded with the interrupt vector and the current address (corresponding with the location of the instruction that is abandoned) is pushed onto the stack. In addition the states of the carry flag (C), the zero flag (Z) and the register bank selection are also pushed onto the stack and further interrupts automatically disabled (IE = 0).



Effects of hardware interrupt when interrupts are enabled (IE = 1)

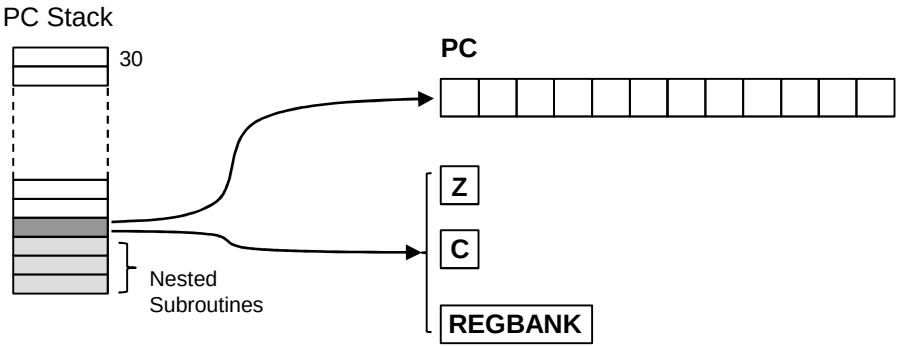
The 'RETURNI' instruction is similar to the unconditional 'RETURN' instruction *but it must only be used to terminate an interrupt service routine* (ISR). When the 'RETURNI' is executed the last address held on the PC Stack is popped off and loaded directly into the program counter so that the program resumes execution starting with the instruction that was abandoned when the interrupt occurred. In addition, the RETURNI restores the values of the carry flag (C), the zero flag (Z) and the register bank selection so that they are exactly the same as when the interrupt occurred. Either the 'ENABLE' or 'DISABLED operand must be used to specify if interrupts are to be enabled or disabled on return from the ISR.

RETURNI ENABLE

IE ← '1'

RETURNI DISABLE

IE ← '0'



Continued on next page….

RETURNI ENABLE
RETURNI DISABLE

See also pages 40-44

Important 1 – Always terminate an ISR with a 'RETURNI' and always terminate a normal subroutine with 'RETURN'. The execution of the inappropriate instruction will result in incorrect operation. Obviously that would be bad enough, but combined with the whole concept of interrupts occurring at any point in the execution of the main code the symptoms of the incorrect operation failure can be subtle and make it extremely difficult to identify the cause.

Important 2 – Just as each 'RETURN' must be executed to correspond with the 'CALL' that invoked a normal subroutine, a 'RETURNI' must only be executed to correspond with the interrupt that invoked the ISR. Your ISR can exploit KCPSM6's ability to implement nested subroutines just as they can be used in any part of your program but it is vital that each level is invoked and completed in order. The maximum number of levels is 30 (but it is rare for this limit to be reached) and it should be remembered that an interrupt used one of them

Examples

```
ISR: ADD sE, 1'd
     ADDCY sF, 0'd
     RETURNI ENABLE
```

This simple ISR increments the 16-bit value contained in the register pair [sF, sE]. This may relate to a scheme in which interrupts occur at regular intervals to provide the base for a real time clock or timer (i.e. the value held in [SF,sE] is then used by the main program when required). The 'RETURNI ENABLE' instruction terminates the ISR and enables interrupts ready for the next time.

```
ISR: INPUT sF, int_data0
     STORE sF, 2A
     INPUT sF, int_data1
     STORE sF, 2B
     LOAD sE, 2A
     RETURNI DISABLE
```

This ISR reads two bytes of information from input ports and stores them in scratch pad memory. It is reasonable to assume that this information relates in some way to the reason for the interrupt and therefore probably represents some important information that had to be captured at that particular time. It can also be imagined that the main program needs to process this special information in some way with the value '2A' loaded into register 'sE' signifying that information has been captured and stored starting at location 2A hex. It can be imagined that the main program must be given time to process the captured information so the 'RETURNI DISABLE' instruction terminates the ISR but prevents a further interrupts overwriting the important information before it has been used. The main program would use an 'ENABLE INTERRUPT' once it had.

```
ISR: LOAD sA 00
     CALL motor_drive
     RETURNI ENABLE
```

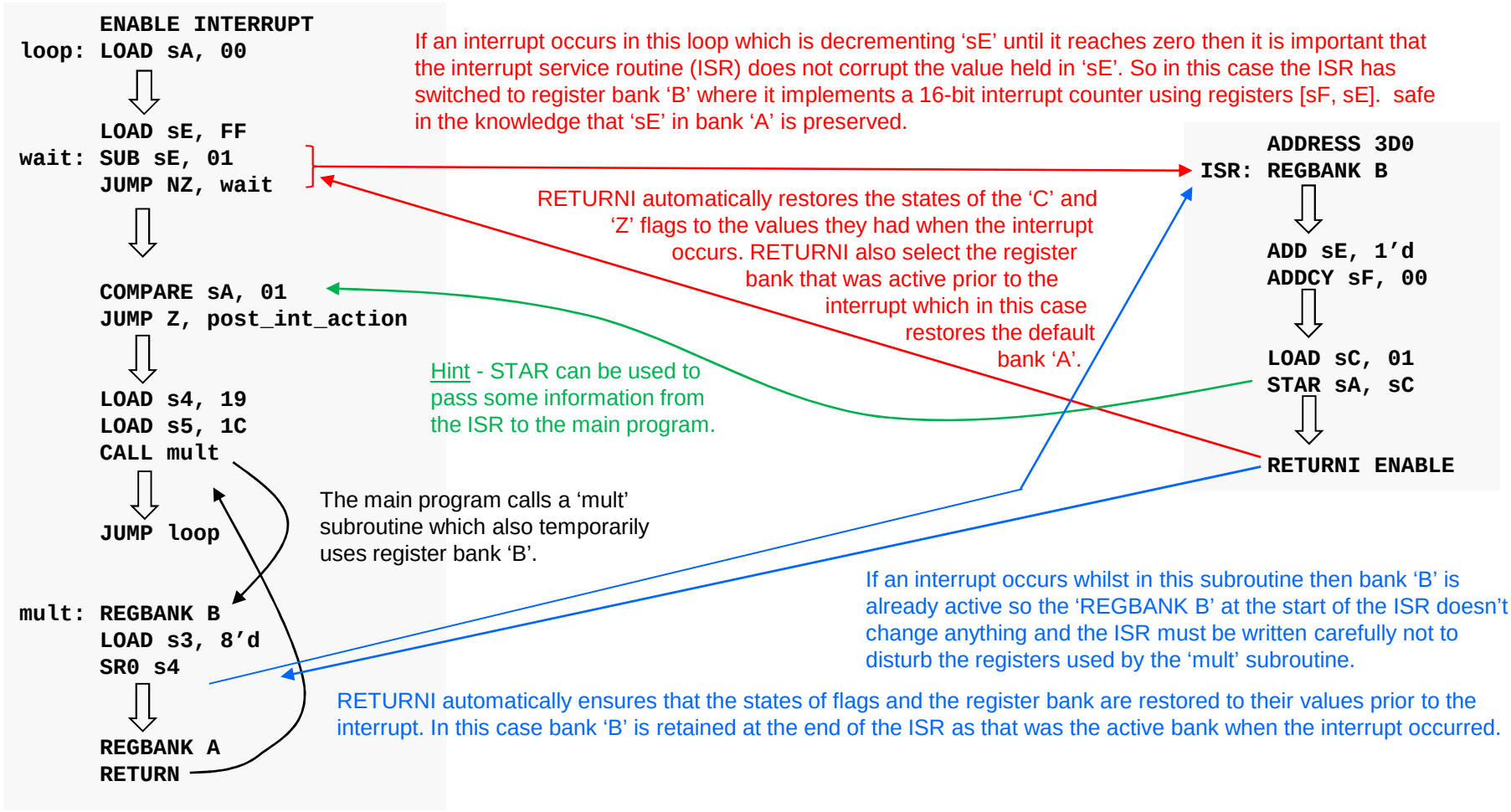
Providing all the normal rules of nested subroutines are followed then an ISR can also make use of subroutines.

```
motor_drive: OUTPUT sA, PWM_value
             OUTPUTK 01, update_strobe
             OUTPUTK 00, update_strobe
             RETURN
```

Hint – Be very careful to make sure that no code executed as part of your ISR procedure contains an 'ENABLE INTERRUPT' instruction.

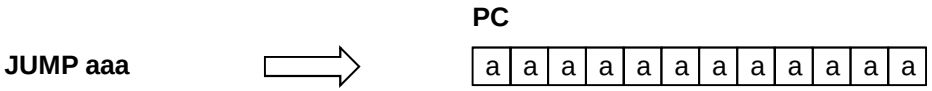
Interrupts and Register Banks

Although both banks of registers can be exploited at any time, the option to reserve at least some of the resisters in the second bank of registers for use in the ISR is very compelling as this guarantees that no undesirable changes are made to the contents of registers used by the main program that has been interrupted. This example illustrates two situations in which an interrupt could occur and how 'RETURNI' always restores the correct register bank.



JUMP aaa

'JUMP aaa' is an unconditional JUMP which forces the program counter (PC) to the absolute address defined by the value 'aaa'. Following power up or a reset KCPSM6 executes code starting at address zero and in general the address increments as each instruction is executed. The unconditional JUMP instruction forces the address to a new value and hence directs KCPSM6 to deviate from the normal incrementing sequence. A JUMP instruction has no effect on any other features within KCPSM6 including the states of the flags.



No change to states of flags.

'aaa' represents a 12-bit address with the range 000 to FFF hex (0'd to 4095'd). However, in nearly all cases it is the assembler which resolves the actual value of the address based on the line labels you include in your program code so you don't have to think about this yourself. Therefore it could be said that the format of the instruction is normally 'JUMP <line_label>'.

Whilst the address range supports programs up to 4K instructions the physical size of the program memory is defined by your hardware. Obviously the larger the program memory, the more block memories (BRAMs) are consumed within the device so the typical methodology is to start with the smallest memory of 1K (address range 000 to 3FF) requiring only a single in a Spartan-6 device and then only to increase the size of the memory if the program outgrows it. It is your responsibility to ensure that the program can fit within the physical address range available but since the assembler resolves most addresses from line labels for you and reports the highest occupied address for your program this is rarely an issue.

Example

```
cold_start: LOAD s0, 27
            STORE s0, status
            ...

warm_start: FETCH s0, status
            OUTPUT s0, LED_port
            CALL update_status
            ...

            JUMP warm_start
```

Nearly all programs include code that is repeatedly executed . These excerpts of a typical program illustrate how a program generally begins with some initialisation tasks that are only performed on power up or following a hardware reset and then the main program code that repeats. The unconditional JUMP forced the execution to loop back to the start of the main program .

The assembler resolves the line label 'warm_start' into an actual address value which you can see in the LOG file if you are interested.

LOG file shows resolved address...

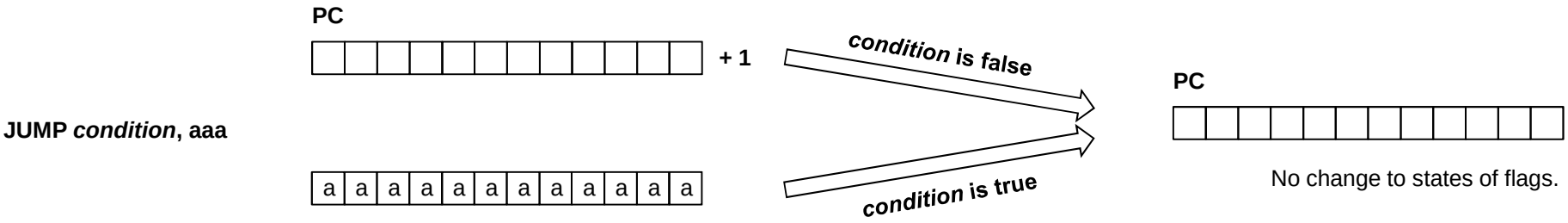
```
32C  2202B  JUMP 02B[warm_start]
```

Hint – The example on the right forces KCPSM6 to stop at the current address. This may seem pointless but a reset or an interrupt can override this and makes KCPSM6 resume a normal execution flow. This technique can be a useful during development and testing of both hardware and software.

```
Halt: JUMP Halt
```

JUMP Z, aaa JUMP NZ, aaa
JUMP C, aaa JUMP NC, aaa

These four conditional JUMP instructions force the program counter (PC) to the absolute address defined by the value ‘aaa’ providing that either the carry flag (C) or the zero flag (Z) is the state specified. If the condition is false the program counter will increment the address and advance to the next instruction. A conditional JUMP instruction has no effect on any other features within KCPSM6 including the states of the flags. See also description of ‘JUMP aaa’.



JUMP Z, aaa JUMP to address ‘aaa’ if the zero flag is set otherwise advance to next instruction.
JUMP NZ, aaa JUMP to address ‘aaa’ if the zero flag is not set otherwise advance to next instruction.
JUMP C, aaa JUMP to address ‘aaa’ if the carry flag is set otherwise advance to next instruction.
JUMP NC, aaa JUMP to address ‘aaa’ if the carry flag is not set otherwise advance to next instruction.

Examples

```
INPUT s0, uart_rx_data
COMPARE s0, "R"
JUMP Z, read_process
COMPARE s0, "W"
JUMP Z, write_process
```

An ASCII character is read from a UART and this is first compared with the letter ‘R’. If the ASCII value is the same then the zero flag is set and hence the flow of the program will transfer to the address associated with the label ‘read_process’. If the character does not match then the program continues and performs a similar comparison with the letter ‘W’ to decide if ‘write_process’ should be executed.

```
          LOAD s0, 25'd
delay100: SUB s0, 1'd
          JUMP NZ, delay100
```

The value contained in ‘s0’ is repeatedly decremented until it reaches zero forming a delay of 100 clock cycles (25 × 2 instructions × 2 clock_cycles). The state of the zero flag is determined by the result of the ‘SUB’ instruction so until the value in ‘s0’ reaches zero the state of the zero flag is not-zero (NZ) so the loop is repeated.

```
TEST sB, FF
JUMP C, odd_parity
AND sB, 7F
```

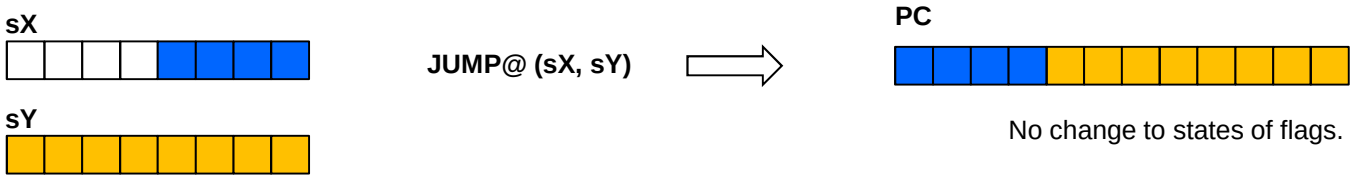
The parity of the 8-bit value contained in ‘sB’ is tested and if it is odd a jump is made to the code located at ‘odd_parity’. When even parity the ‘AND sB, 7F’ instruction is executed.

```
shift: SR1 s3
       JUMP NC, shift
```

Shifts ‘s3’ to the right until a ‘1’ in the least significant bit is shifted into the carry flag.

JUMP@ (sX, sY)

The ‘JUMP@ (sX, sY)’ is an unconditional JUMP which forces the program counter (PC) to the address defined by the contents of the ‘sX’ and ‘sY’ registers.



The 12-bit address is formed of the lower 4-bits of the ‘sX’ register and all 8-bits of the ‘sY’ register. The upper 4-bits of ‘sX’ are ignored and the contents of both registers are unaffected by the operation. There is no restriction on which registers can be used but it would be common coding practice to assign an adjacent pair such as ‘sB’ and ‘sA’.

Since the destination address is defined by the contents of the registers this is powerful instruction but also has the potential to be dangerous! You are entirely responsible for writing a program in which the computed address presented by the pair of registers corresponds with a valid location within your physical program space. The KCPSM6 assembler can do nothing to prevent you computing an inappropriate address but it does provide a mechanism to enable you to determine the addresses associated with line labels as shown in the following example.

Example This example assumes that a user selects an option from a menu by providing a numerical ASCII character in the range “1” to “4” (this range could easily be extended). The program reads this character, converts it to a value in the range 0 to 3 and then jumps to the appropriate routine of ‘choice’.

```
       LOAD sB, menu'upper
       LOAD sA, menu'lower
       INPUT s0, selection_port
       SUB s0, "1"
       ADD sA, s0
       ADDCY sB, 00
       JUMP@ (sB, sA)
menu:  JUMP choice1
       JUMP choice2
       JUMP choice3
       JUMP choice4
```

Without the ‘JUMP@’ instruction the menu would be implemented by a sequential series of compare and jumps (as shown on the right) which does not scale very well but is suitable when there is a small number of choices. Using the ‘JUMP@’ can help when there are lots of choices and also means that the execution time is the same regardless of the selection being made.

```
INPUT s0, selection_port
COMPARE s0, "1"
JUMP Z, choice1
COMPARE s0, "2"
JUMP Z, choice2
COMPARE s0, "3"
JUMP Z, choice3
COMPARE s0, "4"
JUMP Z, choice4
```

The KCPSM6 assembler provides ‘upper and ‘lower attributes that can be used with labels to define the 8-bit constants to be loaded into the registers. These abstracts of the LOG file show how the upper and lower parts of the address are resolved into ‘kk’ values.

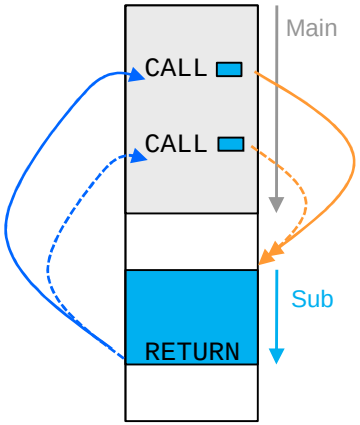
```
7B4  01B07    LOAD sB, 07[menu'upper]
7B5  01ABB    LOAD sA, BB[menu'lower]
```

```
7BB  22862  menu: JUMP 862[choice1]
```

Hint - The ‘upper and ‘lower attributes can also be Used to derive ‘kk’ values for use in other instructions Such as ‘ADD sX, kk’ or ‘COMPARE sX, kk’.

Subroutines

Subroutines are sections of code that are written to perform certain tasks which are then called (invoked) from another section of code. The key point about a subroutine is that it can be called from anywhere within a program and the processor will return to that point in the code when the subroutine's task is complete. This means that it is possible to call the same subroutine from different places within a program because the processor knows which place to return to on each occasion.



An analogy is like you watching a film on DVD and pressing the pause playback button whilst you make a telephone call. Once you have completed your phone call you return to the DVD and press play button to resume the film from the point that you left it. A little tater you need to make another phone call so pause the film again whilst you make that call. Once complete you return to the film to continue the film from the point you left it the second time. In this case the DVD player had remembered where you had paused the film so that each time you returned from making a phone call it resumed playback from the point that you had left it. Making a phone call was effectively a 'subroutine' that you invoked whilst pausing the main task of watching the film.

Reasons for using subroutines.

Common tasks – When the same task needs to be performed at several different places within a program then it is more code efficient to describe it once in a subroutine than replicate in each place that it is required.

Tidy code – By placing the details of particular tasks into subroutines your main code becomes more compact and easier to write/understand.

Code development and Maintenance – A subroutine can be developed and tested in relative isolation until it provides the desired functionality. With good definition of registers, memory locations and ports used by a subroutine the main program will know the 'interface' when calling it and can have a high degree of certainty that it will perform the task expected without unexpected disturbance to register contents etc. As always, the inclusion of accurate and meaningful comments in your code will always be rewarded in the long term.

Library of common tasks – Depending on the application and product there will often be functions that are replicated within a system or carried forward from one generation to the next. These functions will often be specific to your products but they will still be 'common' tasks to you. Well defined and described subroutines make it very easy to copy these functions form a known working design and paste them into your other designs saving time and effort.

Interrupt handling – A hardware driven interrupt is a special case but uses the same mechanism which effectively calls a special subroutine commonly known as an interrupt service routine (ISR) which then returns to the main program at the point at which it was interrupted. This is described in more detail in the interrupt section.

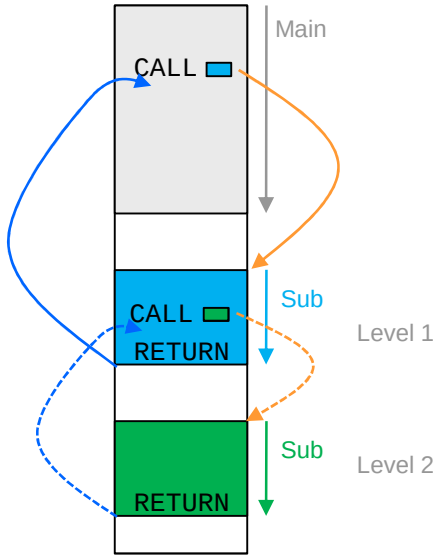
KCPSM6 support for Subroutines

KCPSM6 provides ‘CALL’ and ‘RETURN’ instructions which work with a fully automatic program counter stack which it used to remember the location (address) associated with each ‘CALL’ used to invoke a subroutine in order that it can return to that location when a ‘RETURN’ instruction is executed to terminate the subroutine. There is no requirement for you to reserve any memory, set up any stack pointers or implement any special code, KCPSM6 will do everything for you.

Nested subroutine support

Nested subroutine refers to the ability to call a subroutine whilst already executing another subroutine. The program counter stack within KCPSM6 actually has the ability to support nested subroutine calls to a maximum of 30 levels. Given that it is unusual for a program to exceed 8 levels of nested subroutine calls at one time the 30 levels supported by KCPSM6 provides you with significant freedom when writing your code without the worry of reaching the limit.

It should be remembered that an interrupt is a special case which also uses a level as the ISR is invoked but with 30 levels available even the ISR could include nested subroutine calls of its own.



Your code must be right!

Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that for each CALL made to a subroutine you have a *corresponding* RETURN which completes it. You need to ensure that each CALL and RETURN is a related pair. As the term ‘nesting’ implies, each CALL starts a new level and each RETURN terminates a level in the reverse order. In practice this is all very logical and intuitive providing you truly understand the concept and the diagram on the right.

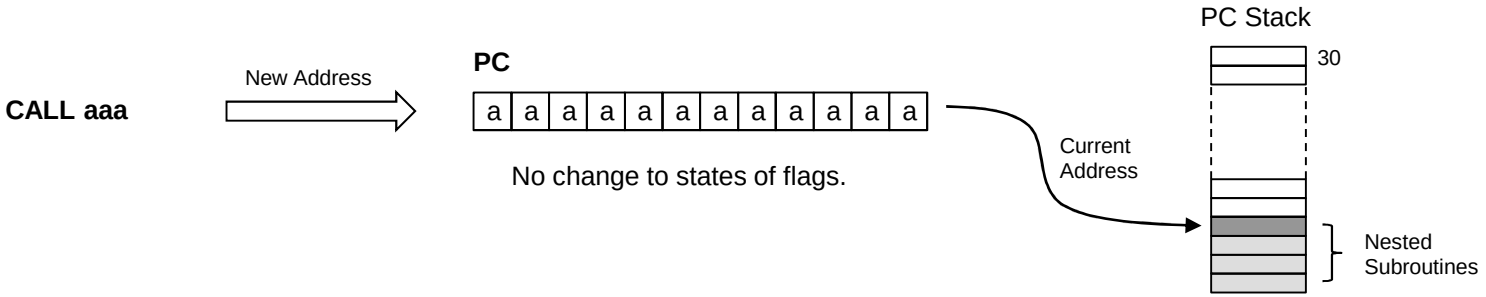
Hint – Although this diagram shows each subroutine located below the other your subroutines can be arranged in any order within the program space and there are absolutely no restrictions on which order in which they can be called. The only thing you need to ensure that a subroutine is only executed by being called otherwise it will encounter a RETURN that did not have a corresponding CALL.

Built-in Protection

In the unlikely event that your program exceeds the 30 levels supported then KCPSM6 will automatically reset. Whilst this is rather dramatic it is predictable when compared with the alternative of incorrect code execution. In practice, the main reason for this every happening is incorrect PSM code where the CALL and RETURN instructions do not correspond. A typical coding error is the use of a JUMP back to a main program instead of a RETURN which can appear to work correctly until the CALL without a corresponding RETURN has executed nearly 30 times leaving inadequate levels for even the correct subroutines.

CALL aaa

'CALL aaa' is an unconditional CALL to a subroutine which pushes the current contents of the program counter (PC) onto the stack and loads the PC with the address defined by the value 'aaa'. A subroutine should end with a 'RETURN' instruction which will pop the last pushed address off of the stack, increment it and load it back into the program counter such that the program then executes the instruction following the initial CALL. Please also see the description of 'JUMP aaa' regarding the valid range of 'aaa' values and how the assembler is typically used to resolve their values for you.



Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that for each CALL made to a subroutine you have a *corresponding* RETURN. You must also ensure that execution of your program does not exceed 30 'nested' subroutines but this limit is rarely challenged by typical programs.

Example Within some code a CALL is made to a subroutine called 'inc_count' which contains a 12-instruction procedure that increments a 32-bit number stored in 4 bytes of scratch pad memory. The corresponding RETURN at the end of the subroutine allows the program to continue.

```
AND s0, 01
OUTPUT s0, status
CALL inc_count32
LOAD s0, 38
JUMP main_loop
```

```
inc_count32: FETCH s0, count0
             FETCH s1, count1
             FETCH s2, count2
             FETCH s3, count3
             ADD s0, 1'd
             ADDCY s1, 00
             ADDCY s2, 00
             ADDCY s3, 00
             STORE s0, count0
             STORE s1, count1
             STORE s2, count2
             STORE s3, count3
             RETURN
```

Hint – A subroutine can be located anywhere in a program relative to the CALL instructions that invoke it but it is vital that the subroutine is only executed as the result of a CALL otherwise their will be no address in the PC stack to correspond with the subsequent RETURN instruction.

CALL Z, aaa CALL NZ, aaa
CALL C, aaa CALL NC, aaa

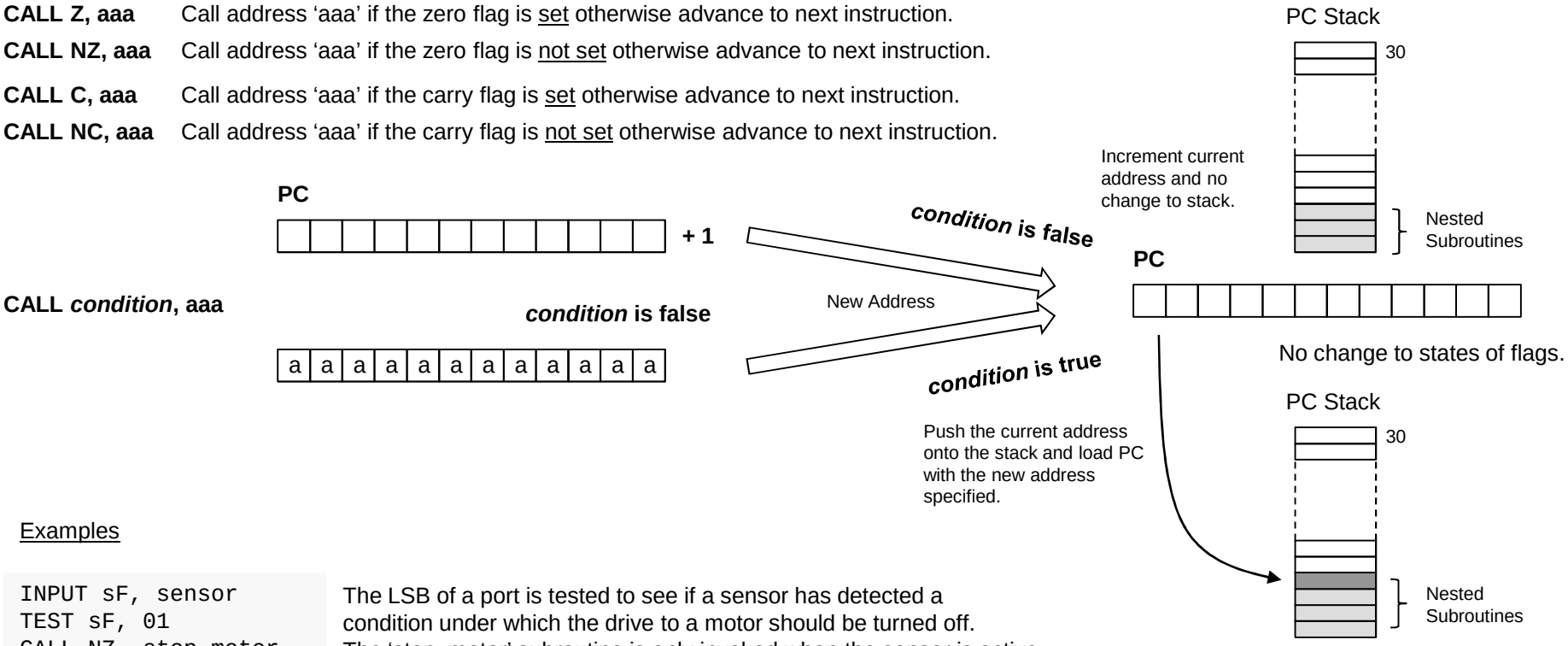
These four conditional CALL instructions invoke a subroutine located at ‘aaa’ providing that either the carry flag (C) or the zero flag (Z) is the state specified. If the condition is false the program counter will increment the address and advance directly to the next instruction . A conditional CALL instruction has no effect on any other features within KCPSM6 including the states of the flags. See also description of ‘CALL aaa’.

CALL Z, aaa Call address ‘aaa’ if the zero flag is set otherwise advance to next instruction.

CALL NZ, aaa Call address ‘aaa’ if the zero flag is not set otherwise advance to next instruction.

CALL C, aaa Call address ‘aaa’ if the carry flag is set otherwise advance to next instruction.

CALL NC, aaa Call address ‘aaa’ if the carry flag is not set otherwise advance to next instruction.



Examples

```
INPUT sF, sensor
TEST sF, 01
CALL NZ, stop_motor
```

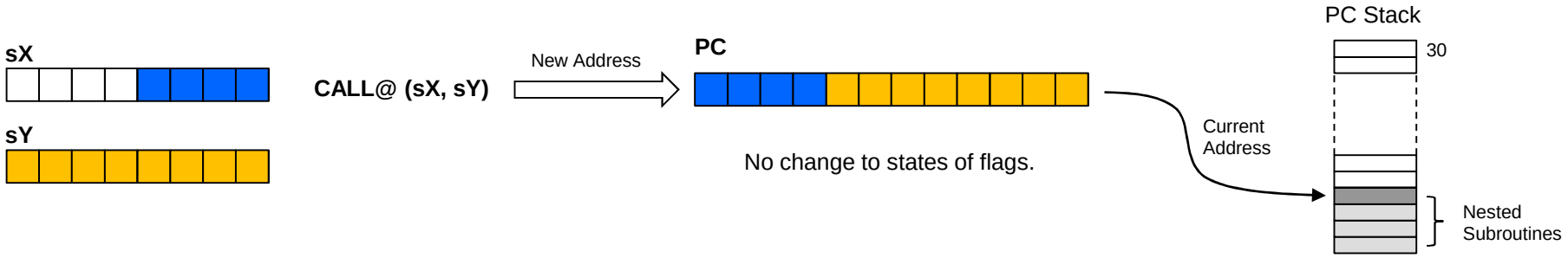
The LSB of a port is tested to see if a sensor has detected a condition under which the drive to a motor should be turned off. The ‘stop_motor’ subroutine is only invoked when the sensor is active.

```
TEST s0, FF
TESTCY s1, 01
CALL C, parity_error
```

An 8-bit value in register ‘s0’ together with a corresponding parity bit located in the LSB of register ‘s1’ are tested. When there are an odd number bits with the value ‘1’ the carry flag is set this is used to detect when there is an even parity error. If that does occur then a special ‘parity_error’ subroutine is invoked.

CALL@ (sX, sY)

The ‘CALL@ (sX, sY)’ is an unconditional CALL to a subroutine which pushes the current contents of the program counter (PC) onto the stack and loads the PC with the address defined by the contents of the ‘sX’ and ‘sY’ registers.



The 12-bit address is formed of the lower 4-bits of the ‘sX’ register and all 8-bits of the ‘sY’ register. The upper 4-bits of ‘sX’ are ignored and the contents of both registers are unaffected by the operation. There is no restriction on which registers can be used but it would be common coding practice to assign an adjacent pair such as ‘sB’ and ‘sA’. As the program counter is loaded, the existing address (the address at which the CALL@ instruction is located) is preserved on the PC Stack to be recovered and used by a RETURN instruction completing the subroutine that has been called.

Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that for each call made to a subroutine you have a corresponding return. You must also ensure that execution of your program does not exceed 30 ‘nested’ subroutines but this limit is rarely challenged by typical programs. The real challenge when using the CALL@ instruction is to ensure that the address defined by the pair of registers does correspond with the start of a valid subroutine. Whilst the instruction facilitates a scheme in which the address to call can be computed this also has the potential to be dangerous! The KCPSM6 assembler can do nothing to prevent you computing an inappropriate address but it does provide ‘upper and ‘lower attributes which are helpful.

Example

Probably the most common use of the CALL@ instruction will be in combination with the LOAD&RETURN instruction to generate text strings or other sequences of constant values and this is explained in more detail in the LOAD&RETURN section.

However, the example shown on the next page illustrates how a system variable is used to decide at which point to enter the same subroutine to achieve a desired set up.

CALL@ (sX, sY)

```
LOAD sB, setup0'upper
LOAD sA, setup0'lower
INPUT s0, currency
SL0 s0
SL0 s0
SL0 s0
ADD sA, s0
ADDCY sB, 00
CALL@ (sB, sA)
```

```
39C  01B0A   LOAD sB, 0A[setup0'upper]
39D  01A49   LOAD sA, 49[setup0'lower]
```

02
04
08
10

49 + 10 = 59

0A + 00 + 0 = 0A

The KCPSM6 assembler provides 'upper and 'lower attributes that can be used with labels to define the 8-bit constants to be loaded into the registers.

CALL A59

```
A49  01047   setup0: LOAD s0, 47["G"]
A4A  01142           LOAD s1, 42["B"]
A4B  01250           LOAD s2, 50["P"]
A4C  016AC           LOAD s6, AC[172'd]
A4D  0170E           LOAD s7, 0E[14'd]
A4E  0184E           LOAD s8, 4E[78'd]
A4F  019BF           LOAD s9, BF[191'd]
A50  22A68           JUMP A68[set_mem]
A51                  ;
A51  01055   setup1: LOAD s0, 55["U"]
A52  01053           LOAD s0, 53["S"]
A53  01244           LOAD s2, 44["D"]
A54  016C3           LOAD s6, C3[195'd]
A55  0172A           LOAD s7, 2A[42'd]
A56  01801           LOAD s8, 01[1'd]
A57  0194A           LOAD s9, 4A[74'd]
A58  22A68           JUMP A68[set_mem]
A59                  ;
A59  01045   setup2: LOAD s0, 45["E"]
A5A  01055           LOAD s0, 55["U"]
A5B  01252           LOAD s2, 52["R"]
A5C  01695           LOAD s6, 95[149'd]
A5D  017C9           LOAD s7, C9[201'd]
A5E  01805           LOAD s8, 05[5'd]
A5F  01911           LOAD s9, 11[17'd]
A60  22A68           JUMP A68[set_mem]
A61                  ;
A61  0104A   setup3: LOAD s0, 4A["J"]
A62  01050           LOAD s0, 50["P"]
A63  01259           LOAD s2, 59["Y"]
A64  016C0           LOAD s6, C0[192'd]
A65  017A8           LOAD s7, A8[168'd]
A66  01831           LOAD s8, 31[49'd]
A67  01920           LOAD s9, 20[32'd]
A68                  ;
A68  2F010  set_mem: STORE s0, 10
A69  2F111           STORE s1, 11
A6A  2F212           STORE s2, 12
A6B  2F63C           STORE s6, 3C
A6C  2F73D           STORE s7, 3D
A6D  2F83E           STORE s8, 3E
A6E  2F93F           STORE s9, 3F
A6F  25000           RETURN
```

8 instructions

8 instructions

8 instructions

In this example we can imagine that KCPSM6 is part of an application involved with foreign currencies and for each different currency it must set up communication using a particular IP Address.

In the main program shown above KCPSM6 reads an input port 'currency' from which it obtains a value in the range 0 to 3 relating to four different currencies but this could be easily expanded to many more. The program then calls the subroutine shown on the right (the LOG file is shown so that the addresses can be seen) which defines the 3-character name of the currency and the IP Address for internet communication in particular scratch pad memory locations.

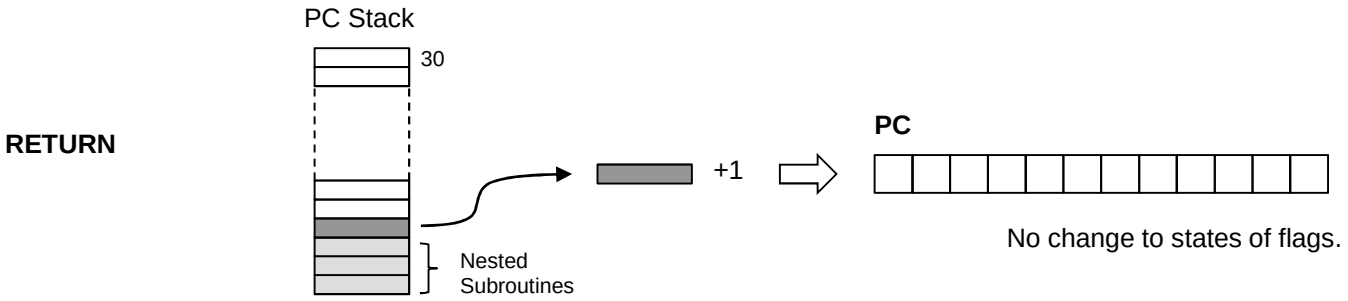
The CALL@ instruction is used to enter the subroutine at the appropriate address to set up the information that corresponds with the value read into 's0' from the 'currency' port. When the currency is 0 then the address is 'A49' and this was known through use of the 'upper and 'lower attributes which were used to load registers [sB, sA] and required no modification.

For currency values 1, 2 and 3 the target addresses are 'A51', 'A59' and 'A61' . Each target address is 8 instructions apart so the value held in [sB, sA] is modified by the addition of the currency values multiplied by 8 (shift left 3 times). The worked example shown in blue shows how currency value '2' is translated into address 'A59' to enter the subroutine at the European 'setup2'.

XILINX

RETURN

The ‘RETURN’ instruction is used to unconditionally complete a subroutine. The last address pushed on to the PC Stack by the previous call to the subroutine is popped off the stack, incremented and loaded into the program counter. This automatic process ensures that the return is made to the address following the CALL instruction that initiated the subroutine.



Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that each RETURN is only executed to complete a subroutine that was invoked by the *corresponding* call instruction. You must also ensure that execution of your program does not exceed 30 ‘nested’ subroutines but this limit is rarely challenged by typical programs. Remember that an interrupt is a special case equivalent to a call and will use one level.

Example

```
LOAD s9, 00
LOAD s8, 00
LOAD s1, 30’d
CALL test_stack
OUTPUT s9, 02
OUTPUT s8, 01

test_stack: ADD s8, s1
            ADDCY s9, 00
            SUB s1, 01
            CALL NZ, test_stack
            RETURN
```

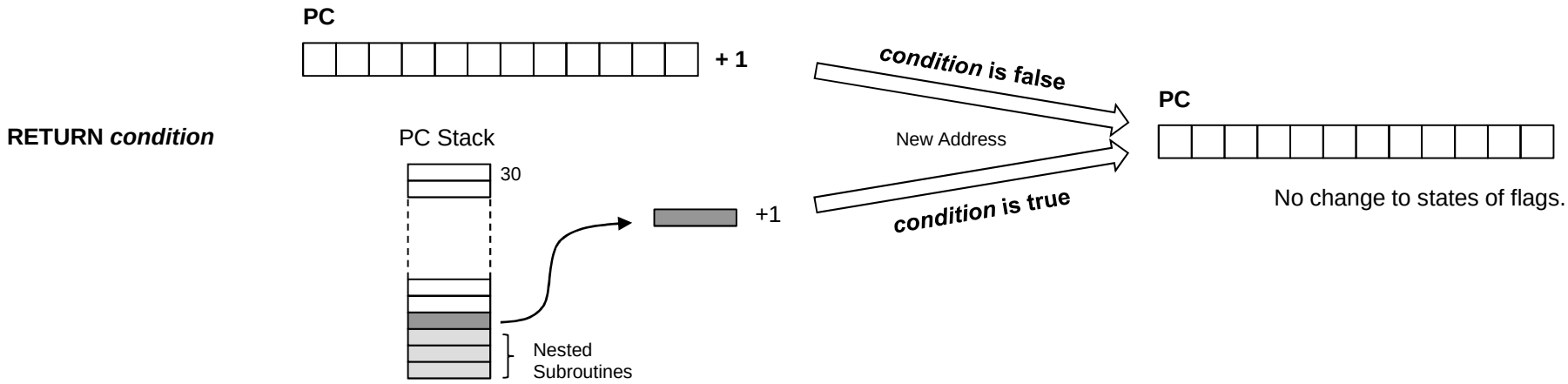
This example illustrates the general arrangement in which one part of the program calls a subroutine. In most cases line labels are used to make the code easier to write and maintain and the assembler resolves the actual addresses.

The subroutine labelled ‘test_stack’ is called from the main program. When this subroutine completes the RETURN forces the program counter to the address corresponding with the instruction immediately following the CALL which in this case is an OUTPUT instruction.

Whilst this example does show the general arrangement it actually describes a rather special case when we look at the code in detail. In the main program [s9,s8] has been cleared and then ‘s1’ has been loaded with 30 decimal. The ‘test_stack’ subrouine adds the value of ‘s1’ to [s9,s8] and then decrements the value in ‘s1’. But each time ‘s1’ is not zero it actually calls ‘test_stack’ again. Hence this subroutine is called 30 times and eventually [s9,s8] will be the sum of all values from 1 to 30 which is 465 (01D1 hex). When ‘s1’ does reach zero, KCPSM6 will execute the RETURN instruction 30 times until it eventually returns to the main program. Hence there is no restriction on how subroutines are arranged providing you do not exceed 30 levels and every CALL has a corresponding RETURN.

RETURN Z RETURN NZ
RETURN C RETURN NC

These four conditional RETURN instructions will complete a subroutine providing that either the carry flag (C) or the zero flag (Z) is the state specified. If the condition is false the program counter will increment the address and advance to the next instruction. See also description of 'RETURN'.



Hint – You are still entirely responsible for making sure that each RETURN is executed to complete a subroutine that was invoked by the *corresponding* call instruction. Because these instructions are conditional you do need to be certain that a corresponding RETURN will be executed at some point so care is required when exploiting these conditional instructions. Many would recommend a coding style in which a subroutine always ends with a single unconditional RETURN instruction and this is certainly a good practice to follow until you have some experience.

Example

```
upper_case: COMPARE s1, 61
            RETURN C
            COMPARE s1, 7B
            RETURN NC
            AND s1, DF
            RETURN
```

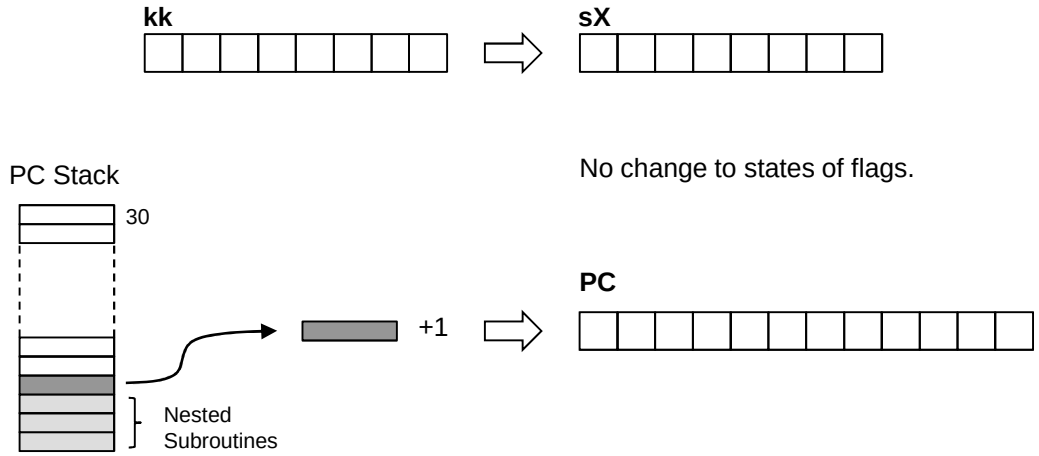
This subroutine converts lower case characters to upper case characters. The subroutine examines an ASCII character code provided in register 's1' to determine if it is a lower case letter in the range 'a' (61 hex) to 'z' (7A hex). If the character falls below or above that range then the conditional 'RETURN C' and 'RETURN NC' instructions are used to terminate the subroutine without any modification to the value in 's1'. When the character falls within the lower case range bit5 of the ASCII code is cleared by the 'AND s1, DF' instruction to convert the code into the range 'A' (41 hex) to 'Z' (5A hex) before the unconditional 'RETURN'.

Note that although this subroutine contains three RETURN instructions one is guaranteed to execute especially as the final one is unconditional.

LOAD&RETURN sX, kk

The ‘LOAD&RETURN sX, kk’ is the combination of a ‘LOAD sX, kk’ and an unconditional RETURN into a single instruction (i.e. one 18-bit instruction that executes in 2 clock cycles). At the same time that the ‘sX’ register is being loaded with any 8-bit constant value, the last address pushed on to the PC Stack is by the previous call to the subroutine is popped back off, incremented and loaded into the program counter.

LOAD&RETURN sX, kk



No change to states of flags.

Example

Probably the most common use of the LOAD&RETURN instruction will be in combination with the CALL@ instruction to generate text strings or other sequences of constant values and this is explained in more detail on the next page. However, the LOAD&RETURN instruction can be used to complete any subroutine with the advantage that an 8-bit value can be loaded into any register at no additional cost.

```
print_decimal: COMPARE s5, 10’d
               JUMP C, convert
               LOAD&RETURN sF, 39
      convert: ADD s5, "0"
               CALL print_character
               RETURN
```

This subroutine is used to convert a value in the range ‘0’ to ‘9’ into the equivalent ASCII character before calling a further subroutine that will print it. In addition to the simple conversion the routine checks that the value provided in ‘s5’ is within the range in order that only the expected ASCII characters are printed (and not nasty control characters etc!). This checking means that the outcome of the ‘print_decimal’ subroutine could be successful or unsuccessful so the LOAD&RETURN instruction is used to load ‘sF’ with a ‘token’ or ‘error code’. If each subroutine set a different error code into ‘sF’ then it would be easy for you or the program to locate where things were going wrong in a program especially during code development.

CALL@ (sX, sY)
LOAD&RETURN sX, kk

"Text Strings"

Hint – Also see TABLE Directive described in 'all_kcpsm6_syntax.psm'.

Using 'CALL@ (sX, sY)' and 'LOAD&RETURN sX, kk' instructions together enables a text strings or sequences of constant values to be generated with maximum code efficiency. The KCPSM6 assembler has a STRING directive that simplifies this common application as shown below.

In this example we will assume that text ASCII characters are output from KCPSM6 to a UART transmitter. The UART macro also contains a 16 character FIFO buffer but given that the serial communication is slow compared with KCPSM6 it is necessary for the program to check that the FIFO is not full before sending another character. The 'send_to_UART' routine on the right will wait until the FIFO is not full before outputting the ASCII character provided in 's1'.

```
send_to_UART: INPUT s0, UART_status_port
              TEST s0, tx_full
              JUMP NZ, send_to_UART
              OUTPUT s1, UART_data_port
              RETURN
```

```
send_Help: LOAD s1, "H"
           CALL send_to UART
           LOAD s1, "e"
           CALL send_to UART
           LOAD s1, "l"
           CALL send_to UART
           LOAD s1, "p"
           RETURN
```

To send a string of characters to the UART you can then repeatedly load 's1' with the next character in the sequence and call the 'send_to_UART' subroutine. This was the fundamental technique used in KCPSM3 programs and it still a valid technique to use with KCPSM6. However as users of KCPSM3 have often reported, this tends to consume a significant amount of code space when there are longer and/or many text strings to be generated. It can be seen in this small example that there are 2 instructions associated with each ASCII character and that is quite an overhead. Although the code can be partly optimised for frequently used characters this does not make code easier to write.

The solution with KCPSM6 is to describe each text string using sequential 'LOAD&RETURN' instructions. In this case 's1' loaded with a different character and this is made much easier to write because of the STRING directive. The original PSM code is shown below and the expanded code is shown in the LOG file on the right.

```
       STRING Hello$, "Hello World"
Hello: LOAD&RETURN s1, Hello$
       LOAD&RETURN s1, 0D
```

KCPSM6

```
3A7                STRING Hello$, "Hello World"
3A7  21148  Hello: LOAD&RETURN s1, 48[Hello$:"H"]
3A8  21165         LOAD&RETURN s1, 65[Hello$:"e"]
3A9  2116C         LOAD&RETURN s1, 6C[Hello$:"l"]
3AA  2116C         LOAD&RETURN s1, 6C[Hello$:"l"]
3AB  2116F         LOAD&RETURN s1, 6F[Hello$:"o"]
3AC  21120         LOAD&RETURN s1, 20[Hello$:" "]
3AD  21157         LOAD&RETURN s1, 57[Hello$:"W"]
3AE  2116F         LOAD&RETURN s1, 6F[Hello$:"o"]
3AF  21172         LOAD&RETURN s1, 72[Hello$:"r"]
3B0  2116C         LOAD&RETURN s1, 6C[Hello$:"l"]
3B1  21164         LOAD&RETURN s1, 64[Hello$:"d"]
3B2  2110D         LOAD&RETURN s1, 0D
```

Each 'LOAD&RETURN' can now be considered to be a single instruction subroutine.

The address of the first 'LOAD&RETURN' instruction is loaded into a pair of registers [sB, sA] and then a routine is called that sends the whole text string to the UART.

```
LOAD sB, Hello'upper
LOAD sA, Hello'lower
CALL send_string
```

```
send_string: CALL@ (sB, sA)
             CALL send_to_UART
             COMPARE s1, 0D
             RETURN Z
             ADD sA, 01
             ADDCY sB, 00
             JUMP send_string
```

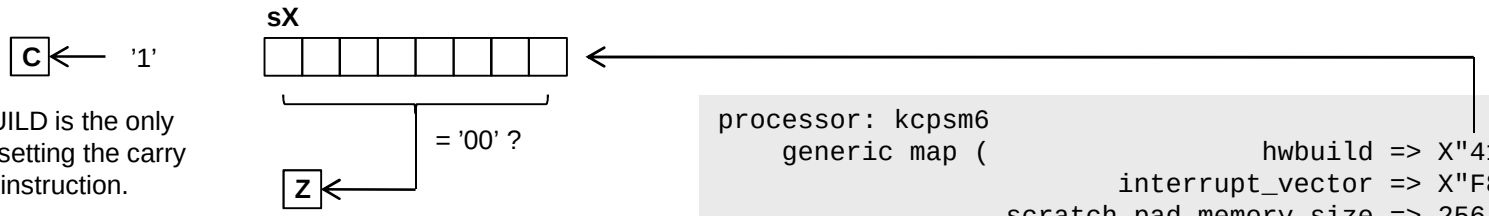
The routine requires a suitable scheme in order to terminate. In this case a carriage return is detected.

The 'CALL@' instruction is used to call each 'LOAD&RETURN' subroutine in turn by incrementing the address held in [sB, sA]. Each call returns a character in 's1' which is then sent to the UART. Although this 'send_string' routine is 7 instructions, text strings are now defined by only one instruction per character improving code efficiency by a factor of two.

XILINX

HWBUILD sX

The ‘HWBUILD’ instruction loads the ‘sX’ register with the 8-bit value defined by the ‘hwbuild’ generic set within the hardware design.



Coding Tip - HWBUILD is the only guaranteed way of setting the carry flag using only one instruction.

```
processor: kcpsm6
    generic map (                 hwbuild => X"41",
                         interrupt_vector => X"F80",
                  scratch_pad_memory_size => 256)
```

The zero flag (Z) will be set if the value loaded is zero and this corresponds with the default value of the generic on the KCPSM6 macro.
The carry flag (C) will always be set (C=1).

Hint – HWBUILD is the only instruction that will always set the carry flag.

You are free to test or use the value loaded into a register by the HWBUILD in any way you wish but here are a few general ways in which you may consider using it in your system…..

Version Control - To enable KCPSM6 to generate a version report (e.g. as part of a message sent to a host or displayed on an LCD). This is the hardware complement to the ‘datestamp$’ and ‘timestamp$’ for version control of the PSM assembly code.

Hint – The value “41” hex shown above could also be used to represent the ASCII letter for version ‘A’ or the packed BCD value meaning version 4.1.

Mode Control - Although most of the functionality would probably be the same, a single PSM file could be written that was capable of different behaviour depending on the hardware in which it was being asked to execute within. For example the HWBUILD could be used to:-

- Define if the unit was to act as a master or slave.
- Adjust the command codes and protocol required to access SPI Flash devices from different manufactures.
- Define the feature set supported and/or included in a product.

Priority or Unit Address - When a unit is placed on a bus then it would generally be assigned an address so that it would know when to respond to commands etc. For example, an audio entertainment system may have multiple speakers that fundamentally operate in the same way but each would be assigned to a particular position in the room and therefore be expected to only generate sounds intended for a particular channel.

Hint – If a system only needs to indentify if it is one of two things (e.g. Left or Right, Standard or Advanced etc) then the ‘hwbuild’ generic could be zero or any non-zero value. When using the ‘HWBUILD sX’ the zero flag (Z) will be set accordingly ready for an immediate decision to be made in the program.