RFFE Command Script Migrator

Ananth

Summary

- Problem statement
- Deliverable
- Spec details
- Test-cases
- Inventory (files, etc)

Problem Statement

- A command script can be used by the simulator to write to (or read from) required registers using the RFFE interface to program chip functions
- The script used for simulation on one project is obsolete for a new project because the addresses to be used (may) have changed
- The information linking chip functions to addresses is contained in an Excel file (NOTE: Unfortunately, the table is not an “Excel table”)
- Using the old and new project Excel files, and a python script, we would like to use the old project script and generate the new script

Deliverable(s)

- Python script
- Confirmation that all tests pass
- OK to assume input files in particular sequence, i.e., :

```
$ mig.py oldREG.xlsx newREG.xlsx old_cmd.txt
```

Specification

- Lines that start with zero-or-more-whitespace, followed by “w” or “ew” will be processed (migrated) (i.e., only “writes” need to be migrated)
- Copy all other lines as is
- Underscore (“_”) in address or data field is for readability only, do not interpret it in any way.
- Output script using underscore to improve readability of data values is desired (that is, even if the input script doesn’t use underscores, please add underscores in the generated output script)
- Line oriented comments (C++ style “//”) to be retained
- Only add comment (chunk name) if original line doesn’t have it

RFFE Writes

```
w, 0101, 0x01, 0000_0001 // write to 5-bit ADDR, USID, ADDR=0x01, DATA=0x01
ew,0101,0x21,0001_0000 // extended write (because ADDR is longer than 5 bits, USID, ADDR, DATA=0x10
```

- How do you know which chip functions are being addressed? A : the “Data Bits” field tells you “which chunk” of the register is being addressed. That is, you see <3> being changed from default value, but <3> is part of <3:0> and so you know that <3:0> is the target
- Now, the “Bit Field Name” column tells you the name of this chunk, which (name) is what is to be used to look for this chunk in the “new” Excel file’s “Register Map Detail” sheet

Write Migration Example 1

```
ew,0101,0x21,01000100
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x11	Reg17[7:4]		BLK_1[3:0]	4b0000
0x11	Reg17[3:0]		BLK_2[3:0]	4b0000
0x21	Reg33[7:5]		BLK_A[2:0]	3b000
0x21	Reg33[4:3]		BLK_B[1:0]	2b00
0x21	Reg33[2:0]		BLK_C[2:0]	3b100

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x31	Reg49[7:4]		BLK_1[3:0]	4b0000
0x31	Reg49[3:0]		BLK_2[3:0]	4b0000

```
w,0101,0x02,00_000_010 // BLK_A
```

- Programming BLK_A because the other “chunks” in REG_ADDR_0x21 stay @ defaults
- BLK_B doesn’t exist on “new”
- Underscores added for readability
- Comment added (note that bus text “[2:0]” dropped)
- Changed from extended write to simple write
- BLK_C data different in migrated version because correct default value needs to be written (i.e., original writes default => migrated should also write default)

Write Migration Example 2

```
w,0101,0x11,00_000_100 // BLK_2[3:0]
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x11	Reg17[7:4]		BLK_1[3:0]	4b0000
0x11	Reg17[3:0]		BLK_2[3:0]	4b0000
0x21	Reg33[7:5]		BLK_A[2:0]	3b000
0x21	Reg33[4:3]		BLK_B[1:0]	2b00
0x21	Reg33[2:0]		BLK_C[2:0]	3b100

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x31	Reg49[7:4]		BLK_1[3:0]	4b0000
0x31	Reg49[3:0]		BLK_2[3:0]	4b0000

```
ew,0101,0x31,0000_0100 // BLK_2[3:0]
```

- Programming BLK_2 because the other “chunks” in REG_ADDR_0x11 stay @ defaults
- Underscores added correctly (generate fresh, ignore underscores on original)
- Comment retained (if generating, it would be “BLK_2”, but there’s already one to keep)
- Changed from simple write to extended write because new ADDR length > 5 bits

Write Migration Example 2 Detail

```
w,0101,0x11,00_000_100 // BLK_2[3:0]
```

- Which address is being written? 0x11
- Which bits of this register are being changed from default? Ans : Only <2> (Using the values available in the “Default” column)
- Which chunk does it belong to? Ans : BLK_2[3:0] (Using info from the “Register Address”, “Default” and “Bit Field Name” fields)
- Which register address contains this chunk in the “new” xlsx? Ans : 0x31 (Using the “Register Address” and “Bit Field Name” fields)
- Migration from “w” to “ew” (or vice-versa) necessary? Ans : Yes – check length of “new” address, if 5-bit or less, “w” else “ew”

Write Migration Example 3

```
w,0101,0x01,0010_0010
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x01	Reg01[7:4]		FN_3[3:0]	4b0000
0x01	Reg01[3:0]		FN_4[3:0]	4b0000
0x02	Reg02[7:0]		unused	8b00000000

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x01	Reg01[7:4]		FN_3[3:0]	4b0000
0x01	Reg01[3:0]		BLK_X[3:0]	4b0000
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x05	Reg05[7:4]		BLK_Y[3:0]	4b0000
0x05	Reg05[3:0]		FN_4[3:0]	4b0000

```
w,0101,0x01,0010_0000 // FN_3
w,0101,0x05,0000_0010 // FN_4
```

- Old script updating FN_3 and FN_4 chunks with a single write – since both chunks have bits set different from default
- But the functions exist in different registers on “new”
- So, two writes are needed (the other way – next example - is trickier – because script needs to remember what it wrote!)

Write Migration Example 4

```
w,0101,0x01,0010_0000 // FN_3
w,0101,0x05,0000_0010 // FN_4
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x01	Reg01[7:4]		FN_3[3:0]	4b0000
0x01	Reg01[3:0]		BLK_X[3:0]	4b0000
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x05	Reg05[7:4]		BLK_Y[3:0]	4b0000
0x05	Reg05[3:0]		FN_4[3:0]	4b0000

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x01	Reg01[7:4]		FN_3[3:0]	4b0000
0x01	Reg01[3:0]		FN_4[3:0]	4b0000
0x02	Reg02[7:0]		unused	8b00000000

```
w,0101,0x01,0010_0000 // FN_3
w,0101,0x01,0010_0010 // FN_4
```

- Do not merge into a single write command
- Python script needs to remember what was written for FN_3 because the intention of the RFFE command script (in this case) is to program both FN_3 and FN_4. So, when writing FN_4, not enough to look at just default for “other chunks” of REG_ADDR_0x01

Lesson of Write Migration Example 4

- In general, after any write, the python script needs to retain the written value to the chunk (needs to maintain a table in RAM)
- The value needs to be used ONLY when the “old” has chunks on different registers that are on the same register in the “new”
- Why, because, if old and new have a register of the form BLK1:BLK2, then we are always updating both BLK1,BLK2 simultaneously
- I.e., first writing 0001_0000 which (say) only updates BLK1 followed by 0000_0001 is specifying to update BOTH BLK1 and BLK2
- The py script sees BLK1 is being written to default value and “concludes” that only BLK2 is being written, but end result is that both BLK1 and BLK2 are update – per user intention

Write Migration Example 5

```
w,0101,0x11,0010_0000
w,0101,0x11,0000_0010
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x11	Reg17[7:4]		BLK_1[3:0]	4b0000
0x11	Reg17[3:0]		BLK_2[3:0]	4b0000
0x21	Reg33[7:5]		BLK_A[2:0]	3b000
0x21	Reg33[4:3]		BLK_B[1:0]	2b00
0x21	Reg33[2:0]		BLK_C[2:0]	3b100

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x31	Reg49[7:4]		BLK_1[3:0]	4b0000
0x31	Reg49[3:0]		BLK_2[3:0]	4b0000

```
ew,0101,0x31,0010_0000 // BLK_1
ew,0101,0x31,0000_0010 // BLK_2
```

- Applying the lesson of Example 4
- If BLK_1,BLK_2 were on different registers in the “old”, then we would have needed to use what was written to BLK_1 from earlier write and use it for the 2nd write here, but since we are always forced to write to both simultaneously, and we see that new value for BLK_2 in 2nd write matches default, even the comment does not indicate that BLK_1 is updated

Write Migration Example 6

```
w,0101,0x11,0010_0000
w,0101,0x11,0100_0010
```

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	4b0000
0x11	Reg17[7:4]		BLK_1[3:0]	4b0000
0x11	Reg17[3:0]		BLK_2[3:0]	4b0000
0x21	Reg33[7:5]		BLK_A[2:0]	3b000
0x21	Reg33[4:3]		BLK_B[1:0]	2b00
0x21	Reg33[2:0]		BLK_C[2:0]	3b100

Register address	Data Bits	C	Bit Field Name	Default
0x00	Reg00[7:4]		FN_1[3:0]	4b0000
0x00	Reg00[3:0]		FN_2[3:0]	**4b1000**
0x02	Reg02[7:6]		unused[1:0]	2b00
0x02	Reg02[4:3]		BLK_C[2:0]	**3b000**
0x02	Reg02[2:0]		BLK_A[2:0]	3b000
0x31	Reg49[7:4]		BLK_1[3:0]	4b0000
0x31	Reg49[3:0]		BLK_2[3:0]	4b0000

```
ew,0101,0x31,0010_0000 // BLK_1
ew,0101,0x31,0100_0010 // BLK_1,BLK_2
```

- Here, in 2nd write, both chunks are being written different from default values, so comment indicates that

Command Line Support

`$ mig.py oldREG.xlsx newREG.xlsx *.txt` should be supported (wildcard in file name)

- Output always goes into the `migrated` (create if it doesn’t exist) directory in the same directory as the source file

I.e., `$mig.py old.xlsx new.xlsx /path/to/x.txt` will result in creation of `/path/to/migrated/x.txt` file (and directory if necessary)

- Ok to overwrite existing file in the `migrated` directory
- Xlsx files can be located in arbitrary directories – assume that command will specify the correct path (name by itself => it’s in cwd)

Inventory of Test Cases

- Unzipping the file gives you a test directory which contains the new and old xlsx files and a few example sub-directories containing input script files (which need to be processed) (and output files, to be used to compare with the output of the python script)
- Run each case from the test directory (i.e., not from within the subdirectory) using the command in the cmd_test file (for ex4 xlsx files are swapped on cmd line, as needed..)
- Then, compare the ex1/migrated/in.txt with ex1/out.txt
- Differences in whitespace can be ignored
- Final example ex8 contains the wildcard filename test

Address Width (Use of `w` or `ew` (Extended..))

- Address is always an 8 bit number, but whether it takes a simple write or an extended write depends on the actual value
- Any value upto and including 31decimal aka 11111 or 0x1F will be a simple write