

# rtR2U2 Quick Reference - Rev. 1

Johannes Geist

July 29, 2013

## 1 Memories

This section will illustrate the memories used within the rtR2U2 framework. A quick overview of the memories with the intended task follows.

### Master Memories

The master has 5 different memories, whereby the duties can be divided in two categories. The Buffer Schedule Memory and the Data Buffer Memory are used for the transmission of results over the bus. The other 3 memories are used for the assessment over the health of the system (adding up all intermediate results and calculating the posterior marginals).

#### Buffer Schedule Memory

- **Name** : `buffer_schedule_mem`
- **Size** :  $\log_2(\text{DATA\_BUF\_SCHEDULE\_MEM\_DEPTH}) \mid \text{DATA\_BUF\_SCHEDULE\_MEM\_ENTRY\_LEN} (\log_2(5) \mid \text{AC\_NODE\_NUM\_LEN} + \text{TRANSFER\_LEVEL\_LEN} + 1 = 16)$
- **Purpose** : This memory contains which results have to be retransmissioned for further calculation. This is needed when an result isn't needed for the next computing level, but for the after next. Currently only one suspended level is supported. **The id of the result as well as the adress of in the `data_buf_mem` is stored.**

#### Data Buffer Memory

- **Name** : `data_buf_mem`
- **Size** :  $\log_2(\text{DATA\_BUF\_MEM\_DEPTH}) \mid \text{DATA\_BUF\_MEM\_ENTRY} (\log_2(5) \mid \text{TRANSFER\_LEVEL\_LEN} + \text{AC\_NODE\_NUM\_LEN} + \text{FP\_PRECISION\_LEN} + 1 = 34)$
- **Purpose** : The actual data which need to be buffered and transferred is stored within this memory.

#### Results Memory

- **Name** : `result_mem`
- **Size** :  $\log_2(\text{RESULT\_MEM\_DEPTH}) \mid \text{FP\_PRECISION\_LEN} (\log_2(32) \mid 18)$
- **Purpose** : This memory has two purposes. The first is to store the intermediate result of the health nodes which are transmitted over the bus. If the health node calculatin is sepereted this memory contains the sum of all this calculations. The second purpose is to hold the posterior marginals of the respective health node. This means that every health node has its own memory entry.

### Results XFER Representation ROM Memory

- **Name** : result\_xfer\_reps\_rom\_mem
- **Size** :  $\log_2(\text{RESULT\_XFER\_REPS\_MEM\_DEPTH}) \mid \text{RESULT\_XFER\_REPS\_MEM\_ENTRY\_LEN}$  ( $\log_2(32) \mid 16$ )
- **Purpose** : If the calculation of a health node over the downward pass is separated this memory encloses how often each health node need to be add up.

### Results XFER Representation Memory

- **Name** : result\_xfer\_reps\_mem
- **Size** :  $\log_2(\text{RESULT\_XFER\_REPS\_MEM\_DEPTH}) \mid \text{RESULT\_XFER\_REPS\_MEM\_ENTRY\_LEN}$  ( $\log_2(32) \mid 16$ )
- **Purpose** : This memory contains the amount of summations still need to be performed

## Computing Block Memories

One computing block contains many small memories. This is due to the used architecture.

### Network Paramter Memory

- **Name** : netw\_param\_ram
- **Entity** : memory\_control
- **Size** :  $\log_2(\text{NET\_PARAM\_MEM\_DEPTH}) \mid \text{FP\_PRECISION\_LEN}$  ( $\log_2(10) \mid 18$ )
- **Purpose** : The probabilities of the Bayes Network are stored in this memory.

### Downward Memory

- **Name** : down\_ram
- **Entity** : memory\_control
- **Size** :  $\log_2(\text{DOWN\_MEM\_DEPTH}) \mid \text{FP\_PRECISION\_LEN}$  ( $\log_2(10) \mid 18$ )
- **Purpose** : Intermediate results, calculated over the upward pass which are needed for the downward pass calculation are stored in this memory.

### Indicator Memory

- **Name** : indicator\_mem
- **Entity** : memory\_control
- **Size** :  $(1 \mid 2^{\text{NUM\_OF\_EVIDENCE\_INDICATORS}})$  ( $1 \mid 2^{20}$ )
- **Purpose** : The updated inputs are stored in this memory. The inputs are the indicator coming from the system under test. Every indicator has 2 bits, one signaling if the indicator is *true* or *false*.

### Instruction Memory

- **Name** : instr\_rom
- **Entity** : instruction\_decoder
- **Size** :  $\log_2 c(\text{INSTR\_MEM\_DEPTH}) \mid \text{INSTR\_LEN}$   
( $\log_2 c(64) \mid \text{AC\_NODE\_NUM\_LEN} + \text{ALU\_LEN} + \text{ALU\_MODE\_LEN} + \text{SAVE\_BITS} + 2 * \text{SINGLE\_OP\_ENTRY\_LEN} = 10 + 2 + 2 + 3 + 2 * (\text{LOC\_LEN} + \text{OP\_ADDRESS\_LEN} = 17 + 2 * (4 + 10) = 45)$ )
- **Purpose** : As the name says this is where the instructions for the computing\_block\_control and the ALU are stored.

### Bus Schedule Memory

- **Name** : bus\_scheduler\_mem
- **Entity** : bus\_attachment
- **Size** :  $\log_2 c(\text{BUS\_SCHEDULE\_MEM\_DEPTH}) \mid \text{AC\_NODE\_NUM\_LEN} + 2 + 4 (\log_2 c(10) \mid 16)$
- **Purpose** : Which data set (AC Number) has to be stored next. This is connected with the bus schedule, which has the corresponding order stored.

### Output FIFO

- **Name** : output\_fifo
- **Entity** : bus\_attachment
- **Size** :  $4 \mid \text{RESULT\_BUS\_WIDTH} (4 \mid \text{AC\_NODE\_NUM\_LEN} + \text{FP\_PRECISION} + 1 + 1 = 30)$
- **Purpose** : If more than one result has to be written on the bus, then this FIFO buffers all the values.

### Dynamic Down RAM

- **Name** : dynamic\_down\_ram
- **Entity** : bus\_attachment
- **Size** :  $\log_2 c(\text{DYN\_DOWN\_MEM\_DEPTH}) \mid \text{FP\_PRECISION\_LEN} (\log_2 c(10) \mid 18)$
- **Purpose** : Some data for the downward pass calculation are calculated on a different computing block. The data needed for the downward pass is marked at the MSB (*true*). This is where memory can be saved, because every dataset is stored in each computing block, even if the dataset is not needed.

### Bus Attachment Memory

- **Name** : bus\_attachment\_mem
- **Entity** : bus\_attachment
- **Size** :  $4 \mid \text{FP\_PRECISION} (4 \mid 18)$
- **Purpose** : The same as the output FIFO but for the inputs. It can happen that more than one input is coming from the bus. If this happens, then the different inputs are stored in this 4 dataset deep memory.

### Register A, B & C

- **Name** : `s_i_reg.a`, `s_i_reg.b` & `s_i_reg.c`
- **Entity** : `computing_block_control`
- **Size** : `1 | FP_PRECISION_LEN (1 | 18)*3`
- **Purpose** : A temporal memory to save data for reload, bus transfer or to store data during the downward pass.

## 2 Procedure of compiling a Bayes network into a binary program for the rtR2U2 framework

The complete compilation process consists out of 5 major steps, which are composed out of several smaller tasks.

- Compile the AC - this is done with the 3<sup>rd</sup> party tool ACE
- Computing Blocks
  - The AC graph is transformed into a  $\Delta$  graph
  - The edges connecting every  $\Delta$  is searched
  - The evidence indicator memory is generated
- Schedule
  - A brute force method is used to obtain an optimized schedule, thereby every  $\Delta$  is shuffled at random and the schedule with the most reloads is kept
  - Thereafter a superior method is used to find a more optimized schedule
- Compile
  - Upward Pass
    - \* Transfer Sources: Find for every  $\Delta$  where the external inputs are from
    - \* Up Blocks: Compile all single instruction lines of each  $\Delta$  and add it to the column of the scheduler matrix
    - \* Instruction Order: Ordering the instructions by a preset tag, during this process the “cycle finish” and “wait for downward pass” are inserted too
    - \* Buffer Memory Schedule: Find out which datasets have to be buffered and create a set of instruction for the reasoning master
    - \* Bus Schedule: Determine which data have to be transferred over the bus
    - \* Bus Attachment Address: ???
    - \* StartUPNOPs: If a computing block is not used in the first round, or the first few rounds then computing blocks can be delayed. **A delay in any other then the first ones is not supported**
    - \* Annotate
  - Downward Pass
    - \* Init: Clear Xfer memory
    - \* compileDOWNcompBlocks: compile the program for every  $\Delta$ , like with the Up-ward pass, everything is written onto the bus
    - \* determinePartialTransfers: find which  $\Delta$  calculates partial results, which are needed for calculation
    - \* optimize for Reloads: find all possible reloads and store this info
    - \* updatePartialTransfers: eliminate bus transfer out of the bus schedule
    - \* assignAddressesToInstructions: The instructions are numbered serially
    - \* trimDownMemoryLoads: change bus transfers to loads out of down memory
    - \* determineSaveToDownMemoryBitsForUpwardPath
    - \* assignDownMemAddresses
    - \* organizeAndCheckBusTransfers: check if all the new created loads are set accordingly
    - \* trimBusTransfers: eliminate the write bus transfers of network parameters
    - \* generateBusScheduleMemoryLayout: determine how many  $\Delta$  calculate one AC number
    - \* generateBufMemSchedule: create the buffer schedule
    - \* assignDynamicDownMemAddresses:

- \* addStartupNOPs: like in the Up-ward pass
- \* annotateInstructions
- \* generateResultXferRepsMemContent: find all evidence indicators and store the ac number as well as the amount of transfers

- Write binary files

An efficient run of the reasoning unit is dependent on the instruction, and efficient instructions are dependent on a “good” AC. Therefore compiling the AC (the first main task) should be performed several times and the best one (narrowest, lowest or something in between (balanced)). (Still needs to be implemented, as well as a complete run of the 5 main task with a single button click and not 5 different ones)

## Compile the AC

One of the simplest task, because it is performed by an external tool. The Bayes network (created in a tool like SamIam), a .net file, needs to be loaded and after that compiled. Currently the best AC has to be chosen manual. As stated above this is still outstanding.

TO DO: A single button to compile the AC in a loop and compile the binary program. There should be also a drop down menu where the "optimization" degree can be chosen. (-o0 : The first One; -o1 : The narrowest out of n; -o2 : The lowest out of n; -o3 : The squarest (a small difference between number of computing blocks and computing levels) out of n)

## Computing Blocks

ComputationGraphBuilder.run

- assignCompBlock (divide the AC in  $\Delta$ )

First the longest distance (distance of each AC node to the root node) is calculated. With this information the AC graph is traversed top to bottom. The first one is added as the TopCompNode (ALU 1) to a computing block. Thereafter possible children are examined. If more than two children are found then the inputs for ALU2 and 3 are used. Otherwise the inputs are set to -1 (non existing). After all AC nodes of one level are visited the AC level distance is incremented by 2 and the loop is started over again.

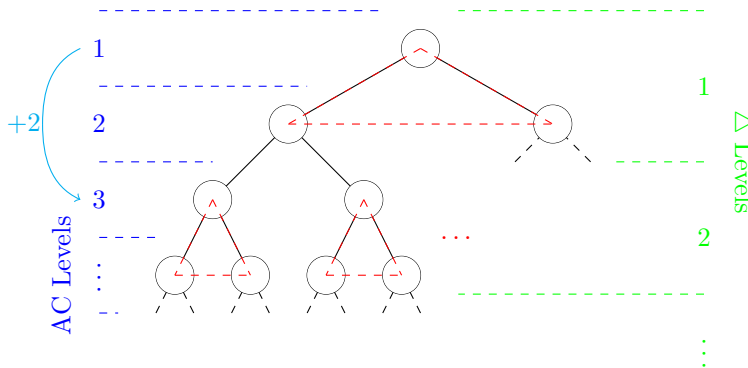


Figure 1: Classification of AC levels and  $\Delta$  levels and the  $\Delta$  itself

After a computing block is filled it is checked that everything really has been added. The last step is to add the new  $\Delta$  to the  $\Delta$  graph.

- simplifyComputingBlockGraph

Due to the mapping process (divide the AC graph into  $\Delta$ ) an overhead is created. This overhead is eliminated in this step. The complete  $\Delta$  graph is traversed and if there is a  $\Delta$  with the same top (ALU1) AC Node number then this  $\Delta$  is dropped.

- **assembleCompBlockGraph (the individual  $\Delta$  are joined together)**

The  $\Delta$  graph is not connected. This function creates the edges between the  $\Delta$ . These edges are then used to create the bus and reload schedule. Then the width (= number of required computing blocks) is calculated. This is done with the distance to the top node; every node with the same distance belongs in the same level. The level with the most  $\Delta$  determines the width of the scheduler matrix and therefore the number of needed computing blocks. **This information can be moved from the current position to the dataManager**

- **assembleLiteralMemory**

The whole AC graph is gone over and every AC node, which is a evidence indicator (also called literal) is put into a separate ArrayList. The indicator list is then transferred into a memory. This memory is the same in every computing block. **The memory size can be reduced if only the evidence indicators needed for each node are stored.**

- **Sanity Check**

## Schedule

### **.run method**

The idea is to find the best filling for the scheduler matrix. The scheduler matrix is as wide as defined by the  $\Delta$  graph ( $\Delta$  level with the most  $\Delta$ ). The height is defined by the  $\Delta$  levels. The first one is random and the goal is to find a schedule with maximum reloads (1 reload per  $\Delta$  is the maximum). This leads to fewer bus transfers, which need more computing time.

- root node is at (0,0) - (x,y) - (left top)
- a  $\Delta$  can be fixated e.g. root node
- first a brute-force approach is used and after that a more intelligent method is applied

### **Flow of optimization:**

The  $\Delta$  of every row are shuffled, row after row. After this process the current reloads are counted. If the current reloads are more than the current best schedule has then the current schedule is stored. This is done for a predefined number of seconds.

The superior method looks if a swap of two  $\Delta$  increases the number of reloads. This is done for the whole schedule matrix.

*The brute-force process is superior for small ACs and the intelligent way for bigger ones.*

## Compile (Transfers = Bus)

This step transfers the  $\Delta$  graph, with the information of the scheduler matrix into the binary program.

### Upward pass

#### **.run method**

The main steps are visible in the run method. As all run methods this one is invoked with the dataManager too.

- TransferSources

Find the source of the external inputs of every  $\Delta$ . This first step determines which datasets can be reloaded and which ones coming from the bus.

- compileUpBlocks

Compose every single instruction of every  $\Delta$ . In this step the network parameter memory is put together too. Flow of Compilation:

- check if there is a ALU2 or 3
- check if data is reloadable
- compile ALU2 and 3
- compile ALU1
- add line to instruction and add a number too

With the information of the scheduler matrix it is determined which  $\Delta$  instructions come together. All instructions of one column (schedule matrix) are thrown together.

- orderInstructions

Order the instructions according the assigned numbers. Sorted list is put into the computing block list. The instructions for *cycle end*, as well as *waiting for downward pass* are added (at the end of one complete run). The top node, the result of the upward pass is identified (all 1s (AC node number)).

- generateBufMemSchedule
  - go through the  $\Delta$  graph bottom to top
  - if the result of one  $\Delta$  is needed in a level + 2 then this results needs to be buffers
  - this information is created in this step and stored in the master
- generateBusSchedule
  - cycle through all external inputs (inputs of  $\Delta$ )
  - find all evidence indicators and network parameters, as well as reloads
  - eliminate these
  - find the node (external input) with the remaining id within current level - 1 and add these to the schedule
  - all others have to be buffers and transmitted over the bus in a later round (coming from master)
  - apply a sanity check
- assignBusAttachmentAddress

More than 1 input can be from the bus, therefore there is 4 entries deep memory. Every bus load is referenced with and id(0 to 3), which are also the memory addresses. This is possible, with the information of the bus schedule. Every single bus read is rewritten with the corresponding read from the bus attachment memory entry.

- addStartUpNOPs

Not every computing block starts in the first level. Therefore these computing blocks have to stall. This is done by appending NOPs prior to the other instructions. NOPs don't have to be appended after the instructions, because there is the *wait for DW pass* instruction.



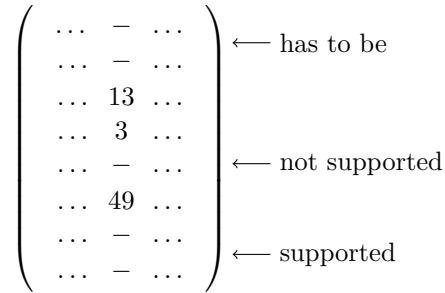


Figure 2: Scheduler matrix with supported NOP levels.

- annotateInstr

Only for debugging. Just to understand the binary code better.

### Downward pass

- initialize

The only task performed is a clear of the resultXferRepsMem object. **As far as known this is only necessary for a rerun of the compile procedure.**

- compileDOWNcompBlocks

The upwardpass schedule matrix is used to determine where every  $\Delta$  is calculated. Thereafter, the instruction of one  $\Delta$  is created. The corresponding method (compileBlock) create an initial program were every operand is either loaded from the register C, the downward pass memory or the network parameter memory. Every result of the top node of a  $\Delta$  is written onto the bus and only the intermediate results are stored in the Register A and B.

- determinePartialTransfers

Not every data set can be loaded from the register or down memory, as it is assumed on the step before. Therefore, in this step all results of a level above are searched for. Reading, every edge coming from one  $\Delta$  and connecting to a  $\Delta$  a level below.

- optimizeForReloads

Go through one column of the scheduler matrix and find if there is any  $\Delta$  input which is a result of another  $\Delta$ . If this is the case, then redirect the result to the register C and change the location of every other input within the column of the scheduler to register C. This step does not look into other columns, therefore if the result is needed by another hardware block this will taken care of another step.

- updatePartialTransfers

Get rid of the entry in the transfer sources list, because the results are reloaded within the hardware block.

- assignAddressesToInstructions

Number the instructions all the way through, starting by 0.

- trimDownMemoryLoads

Change the default (downward memory) to network parameter or evidence indicator memory, if possible. Also find the corresponding address in the memory and change the numbered instructions.

- determineSaveToDownMemoryBitsForUpwardPath

Go through all downward pass instructions and find every load from the down memory. Afterwards go through the upward pass instructions and compare the AC node numbers. If the AC node number matches set the “save for downward pass” bit. Furthermore, add the value to the down memory (the AC node number as an ID) and increase the address.

- assignDownMemAddresses

Connect the newly created down memory with the downward pass. The addresses of the downward pass are initialized with a placeholder (ZERO\_REG\_ADDRESS) and this step replaces the placeholder with the actual memory address.

- organizeAndCheckBusTransfers
- trimBusTransfers

Get rid of all the bus transfers of network parameters, as we are not interested in these results. Iterate over all downward instructions and compare the AC node number with the network parameter numbers, if they match set the save bits to SAVE\_INFO\_NONE.

- generateBusScheduleMemoryLayout

Create the schedule for bus transfers. This is done by going through all  $\Delta$  and store how often the AC node is calculated and transmitted. BUG: if the same AC node result is interrupted by another result than the interrupting result is dismissed.

- generateBufMemSchedule

Not all intermediate results of a node are calculated in the same level therefore some results have to be stored and retransmitted in another level.

- assignDynamicDownMemAddresses

Some data set for the downward pass are calculated on a different hardware block, therefore these data sets have to be stored during the upward pass. The procedure is as followed. Go through all instructions of the upwardpass and if the instruction has the same AC node number as the missing data set of the downwardpass than rewrite the instruction. The instructions’ AC node number is increased by a defined offset (512) and inserted back to the program. Furthermore, the new created Instructions are connected with a new location (dynamic down memory) and associated address.

- addStartupNOPs

The same procedure as with the upwardpass is used.

- annotateInstructions

For better reading the Instructions will be layouted and commented, these information are only for debugging and control purpose. It will not be transferred to the Hardware.

- generateResultXferRepsMemContent

All evidence indicators are health nodes. Therefore all intermediate results transferred during the downward pass are essential. All instructions are passed through and if the result is written onto the bus and is a evidence indicator than the AC node number is stored in a memory. Furthermore, if the calculation is distributed on more then one computing block, than a summation counter is stored too. For a simpler hardware implementation a lower and upper bound AC node number is stored and all evidence indicators are within these bounds.

## Write binary files

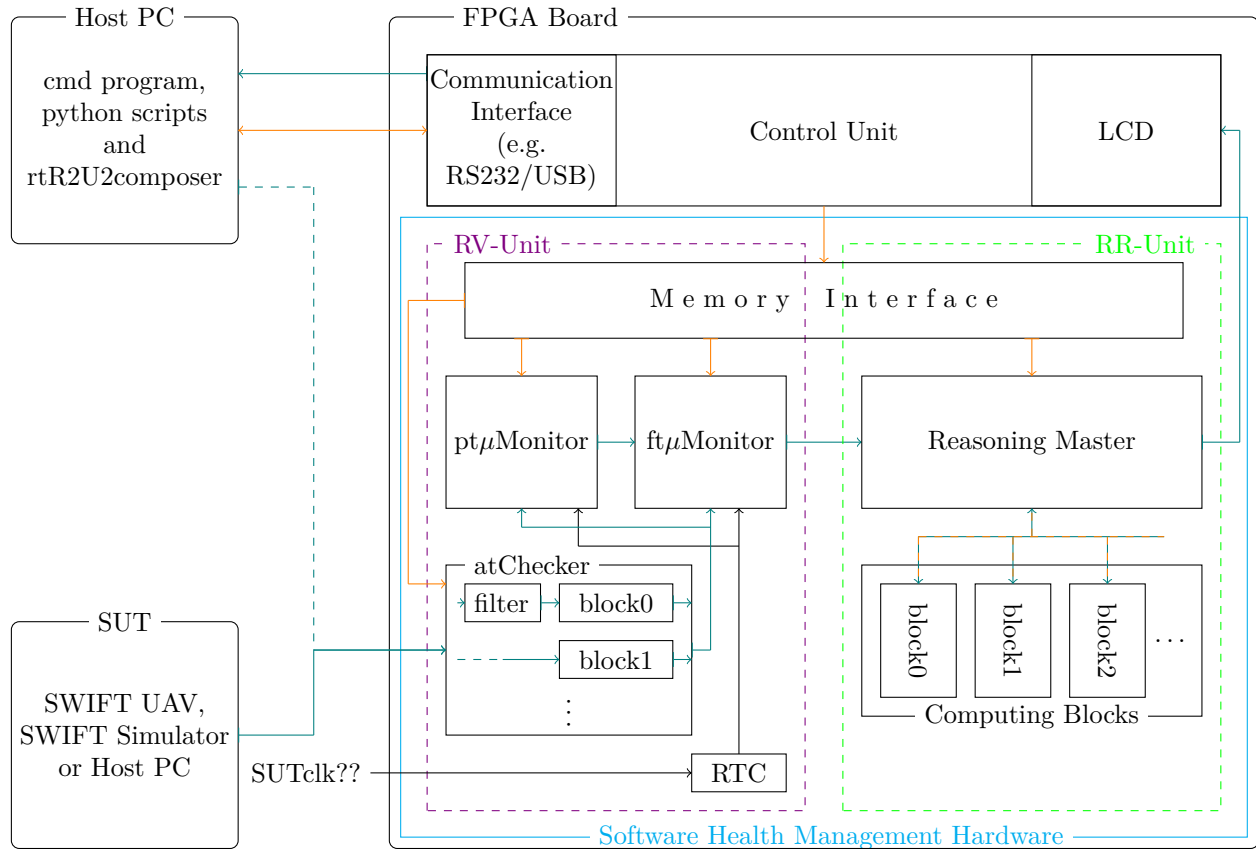
The created abstract data is transformed into binary files, which can be transferred to the Hardware. This is done with predefined instructionset templates and toBinary methods.

### **3 Preparation**

This section describes all necessary preparation steps to create the binary file.

## 4 Scratchpad

- visited flag signals if this AC Node has been visited e.g. tree traversal
- dataManager maintains nearly all data for the reasoning compile task
- reset data!! this is not 100% implemented! Needs a check; needed for several iterations of the compile process





-  running data
-  programming data

Figure 3: An Overview of the framework, with data flow.