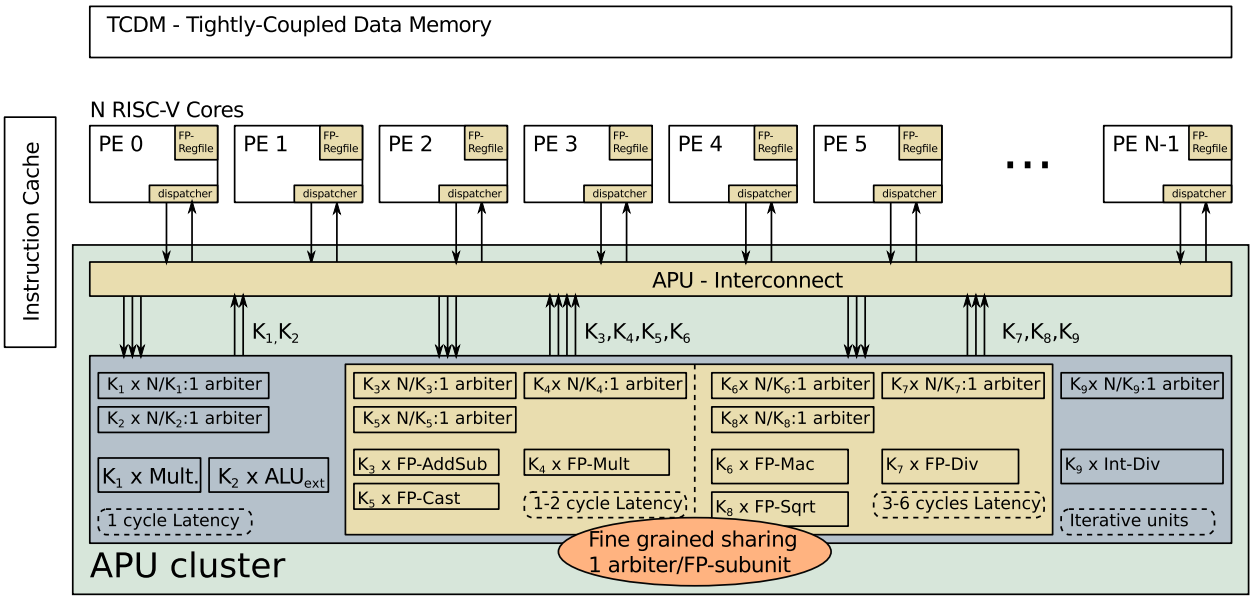
A Shared Auxiliary Processing Units



Michael Gautschi
gautschi@iis.ee.ethz.ch

July 4, 2017

Contents

1 Overview

The shared auxilary processing unit (APU) is designed to work with the RI5CY core and aims on sharing expensive ISA extensions in one shared unit among all cores. RI5CY is a four-stage in-order RISC-V core architecture which supports the following instructions:

- Full support for RV32I Base Integer Instruction Set
- Full support for RV32C Standard Extension for Compressed Instructions
- Full support for RV32M Standard Extension for Integer Multiplication and Division
- PULP specific extensions
  - Post-Incrementing load and stores
  - Multiply-Accumulate and dot-product extensions
  - ALU extensions
  - Hardware Loops
- Full support for RV32F Floating Point Extension through a shared or a private FPU

This document describes how the shared APU can be used, to better amortize costly instruction extensions, such as the full floating-point RV32F extension. Chapter 2 will detail the architecture of the shared APU including the building blocks such as arbiters, interconnects, and the shared units. Chapter 3 explains the required core extensions for interfacing the shared APU, and Chapter 4 will provide information about the current functionality of the shared APU, mainly the shared dot-product instructions, and the shared FPU.

1.1 Overall Sharing Concept

The idea of sharing units is that all cores in a multi-core cluster see the shared resources as if they had private instances. Hence, to a programmer, the sharing of resources is completely transparant. There are two distinct ideas on sharing resources as indicated in Figure 1.1. We differentiate between a `simple sharing` and a `fine-grained sharing`. The former shares one block which itself supports different operations. This approach is typically prefered when the different subunits are not used concurrently, and they can itself share a large part of its datapath. The logarithmic number unit (LNU) [1] is a good example for this sharing approach, because all operations are interpolated in a second order interpolator which is in common for all operations, but the look-up tables are different distinct.

The fine-grained sharing is more complex and does not aim to share a single unit, but directly the individual operators. This has several advantages because it allows for a fine-grained adjustment of unit instantiations (e.g. one FP-divider and 2 FP-FMA units), it allows the different types

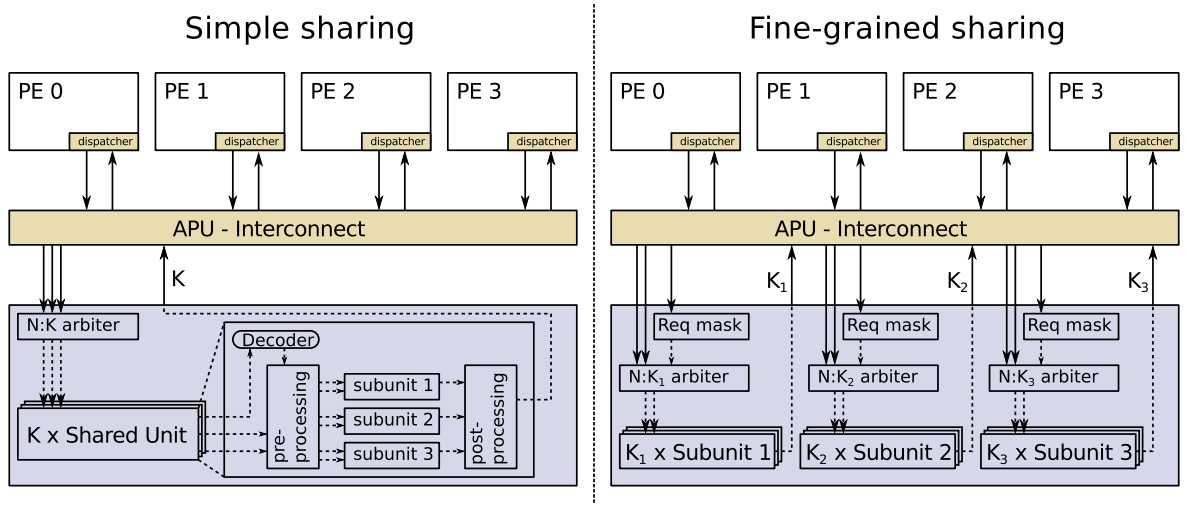


Figure 1.1: Simple sharing: One type of unit with support of different subfunctions is shared. Fine-grained sharing: the subfunctions are shared directly.

of units to have a different latency (e.g. multi-cycle FP-divisions, and two-cycle FP-additions), and it allows the cores to access the different units concurrently. In contrast to just instantiating multiple units, this allows to better control the number of subunits, minimize the overall area consumption, and therefore use a simpler arbitration circuit (an 8:1 arbitration is faster than an 8:2 arbitration).

Supporting different latencies in the fine-grained shared unit is only possible if the interconnect can stall the different units, and cores to resolve access and write-back conflicts or if the cores know the latencies of the individual instructions and can avoid collisions before they occur. The first approach is a bit more flexible but poses additional constraints on the interconnect. This makes the interconnect more complex, but more importantly, increases the latency of the interconnect which forces to use additional pipeline registers in the units itself leading to more stalls.

For this reason, we have opted for the second option where each core is aware of the instruction latencies and only issues new instructions to the APU which have a latency which is greater or equal to the latency of the previous instruction.

In the following the fine-grained shared auxiliary processing unit (APU), the interconnect and the core dispatcher will be explained in more detail.

2 The APU Architecture

This chapter deals with the interconnect in Section 2.1, the shared subunits in Section 2.2, and the interface of the APU to the cores in Section 2.3.

2.1 Interconnect

As shown in Figure 2.1 the task of the interconnect is to connect N PEs with all L types of shared execution units. Each dispatcher signals, that it wants to offload an instruction with a `request` signal. A `type` signal is used to route the instruction which consists of `operands`, `opcode`, and `flags` to the corresponding unit. For each type L the `type`, `request`, `operand`, and `opcode` signals of all N dispatchers have to be replicated and routed to the shared units. Each shared unit can execute one operation per PE, and hence generates N `grant`, `result`, and `valid` signals which have to be routed back to the N PEs. Since each PE can only issue one transaction per clock cycle, the reduction block for the `grant` is a simple logic *or* of all L types. If all shared units have the same latency, it is not possible that more than one result is valid at the same time. Hence, only one result needs to be selected with a $L:1$ multiplexer, and forwarded to the core. Since it is unlikely that all shared units have the same latency, additional constraints are posed on the dispatcher to make sure that transactions of a given PE do not overtake each other or finish concurrently which would mean that multiple results would have to be routed back to the same PE in a single clock cycle. These constraints are explained in the following in more detail.

2.2 Arbitration and Shared Units

To support a fine-grained sharing of execution units as shown in Figure 1.1, each type requires its own arbitration unit which assigns the K_i units according to the N requests. Each arbiter is identified with a unique type identifier and the requests are filtered by the type before the actual arbitration is performed. A simple round-robin arbiter has been used for the arbitration of N cores and K_i shared units. The arbiter generates a tag that is used to identify the source. This tag is then forwarded to the `input_sel` unit to select the right input operands, and to the actual shared unit which can be implemented as

- an iterative, blocking unit
- or a pipelined unit.

Figure 2.1 shows a pipelined unit with x registers stages which can accept one operation per cycle, and outputs the result after $x+1$ cycles. In this case, the tag needs to propagate through a separate pipeline of the same depth. The tag can then be used in the `output_sel` unit to select the right result and forward it to the right core. Note, because all K_i units of one specific

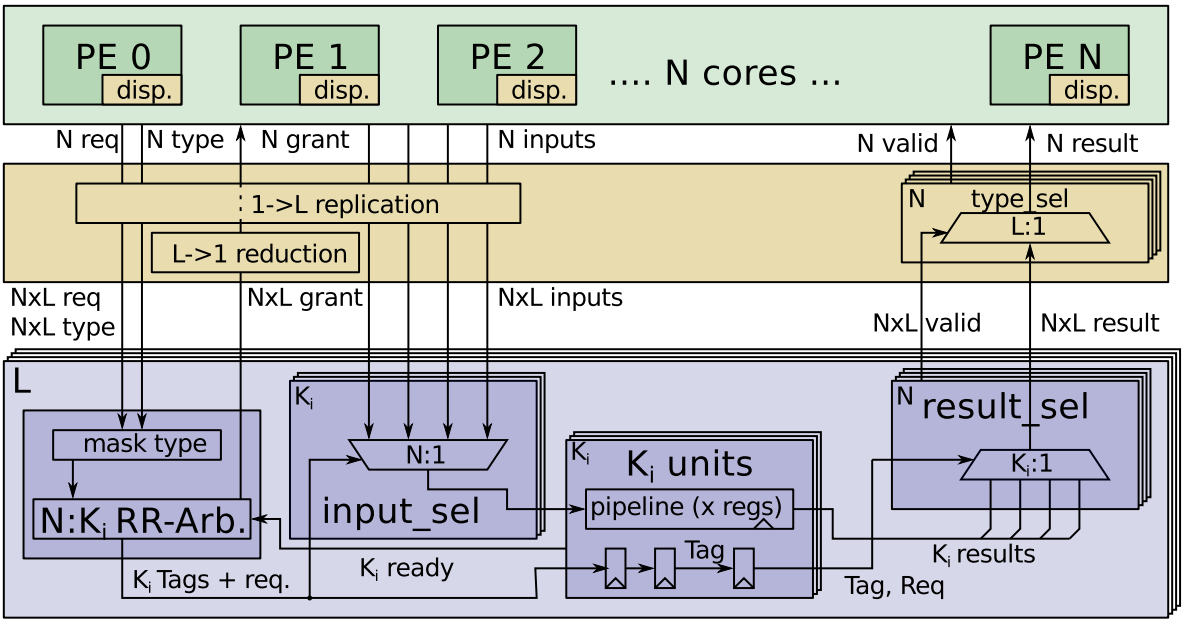


Figure 2.1: Interconnect with multiple types of execution units.

type have the same latency, and a core can only issue one operation per cycle, it is not possible that two shared units of the same type have a result for the same core. Therefore, a simple multiplexer can be used to select the output.

Iterative, blocking units are also supported, but require an additional `ready` signal to inform the arbiter whether is ready to accept a new operation or not. This `ready` signal is used by the arbiter to decline requests to this specific unit.

Cluster implementations with more than four cores likely require sharing more than one unit of each type without significantly increasing the number of access contentions. The more cores (N), and shared execution units (K_i), the more complex the arbitration, and input/output selection becomes. In fact, for such cases, a round-robin arbiter might not always be the preferred solution because its delay is too high and would lead to a frequency degradation.

The units can have different latencies and we distinguish between three groups which are:

- single-cycle units (combinational units) (1 cycle delay)
- two-cycle units (2 cycles delay)
- multi-cycle and iterative units (more than two cycles delay)

The core has to know to which group each instruction belongs in order to issue instructions correctly.

Table 2.1: APU Interface Signals

Signal Name	Direction	Width	Explanation
clk_i	input	1	clock
rst_ni	input	1	reset
Handshake signals (one instance per core)			
cpus[i].req_ds_s	input	1	request signal of core i
cpus[i].type_ds_d	input	clog2(NTYPE)	type signal of core i to identify the type of units it wants to access
cpus[i].ack_ds_s	output	1	grant signal from arbiter to core i to accept a request
cpus[i].valid_us_s	output	1	valid signal for core i
Data signals (one instance per core)			
cpus[i].operands_ds_d	input	3x32	the three FP- or INT-operands of core i
cpus[i].op_ds_d	input	NOP	opcode of the operation of core i
cpus[i].flags_ds_d	input	NDSFLAGS	flags for the shared units of core i
cpus[i].ready_us_s	input	1	ready signal of core i (not used statically set to 1)
cpus[i].result_us_d	output	32	result of the shared unit for core i
cpus[i].flags_us_d	output	NUSFLAGS	flags from the shared unit for core i

2.3 APU Interface

The interface signals of the APU are described in Table 2.1. Each core is sending one instance of the interface type `cpu_marx_if()` to the APU. The handshake consists of a request, a grant, type, and valid signal. The core sends a request and holds it high as long as the shared unit answers with a grant. During this time the type signal shall not change. In the cycle in which the grant is sent to the core, the data input signals are captured and forwarded to the corresponding unit. When no contentions occur, this all happens in the same clock cycle. Depending on the latency of the shared unit, the valid signal goes high and signals that the result and the flags are valid and can be forwarded to the core. The interconnect has no possibility to stall the write-back process because the core dispatcher is always ready to receive a result.

The widths of the type signal depends on the number of different types of shared units (NTYPE). The width of the opcode is the maximum opcode width of any shared unit. Similarly, NDSFLAGS and NUSFLAGS determine the number of up-, and down-stream flags.

3 Core-Extensions

This chapter introduces the additional circuits in the core architecture to support shared execution units. First, the instruction decoder is explained in Section 3.1 and Section 3.2 deals with the required instruction dispatcher that guarantees in-order execution. Finally, the core interface and the available performance counters are explained in Section 3.3 and Section 3.4.

3.1 Instruction Decoding

The decision to offload an instruction to the shared unit is generated in the instruction decoder, which is located in ID-stage. If the right signals are set, the ID-stage is then forwarding the operands, opcode, and possible flags to the APU dispatcher, rather than to the ALU, multiplier or private FPU. The decoder needs to set an enable signal to activate the dispatcher, the opcode and the type of the shared unit for this specific operation and finally the instruction latency which can either be one, two or three. Whereas one means that the shared unit is a combinational block of latency one, two means that the unit has one pipeline stage and a latency of two cycles, and three includes all other shared units with a latency greater or equal to two cycles. Further, the assigned type signals have to match the ones of the APU. Hence, it is not enough to redefine a type signal in the instruction decoder, but it also has to be changed accordingly in the APU.

3.2 APU Dispatcher

The APU dispatcher is a small unit located in the EX-stage that is capable of offloading operations to the shared unit, and at the same time, handling access contentions, checking for data hazards, and write-back contentions with private execution units. To make sure that transactions in the shared units do not overtake each other, or finish at the same time causing a write-back contention, each instruction has been annotate with a latency of one, two, or many cycles. This allows the dispatcher to know instead of speculate whether it can issue more instructions in the next cycle or needs to stall to avoid contentions. If the dispatcher issued a one or two-cycle operation, it can issue more instructions because the pipeline can hide latencies up to two cycles by using the two WB ports. Single-cycle operations can be treated as ALU operations and can write-back on the first WB port which is already used by the ALU. Two-cycle operations on the other hand are writing back on the second WB port which is also used by the LSU when loading bytes from memory. Finally, three and more cycle instructions will stall the pipeline to avoid having too many outstanding transactions.

Figure 3.1 shows six different scenarios that show what instruction sequences will force the dispatcher to stall the pipeline. The notation `APU 2` means that the instruction will be executed on the APU and will take two cycles and `wb 2` means that this instruction is written back on

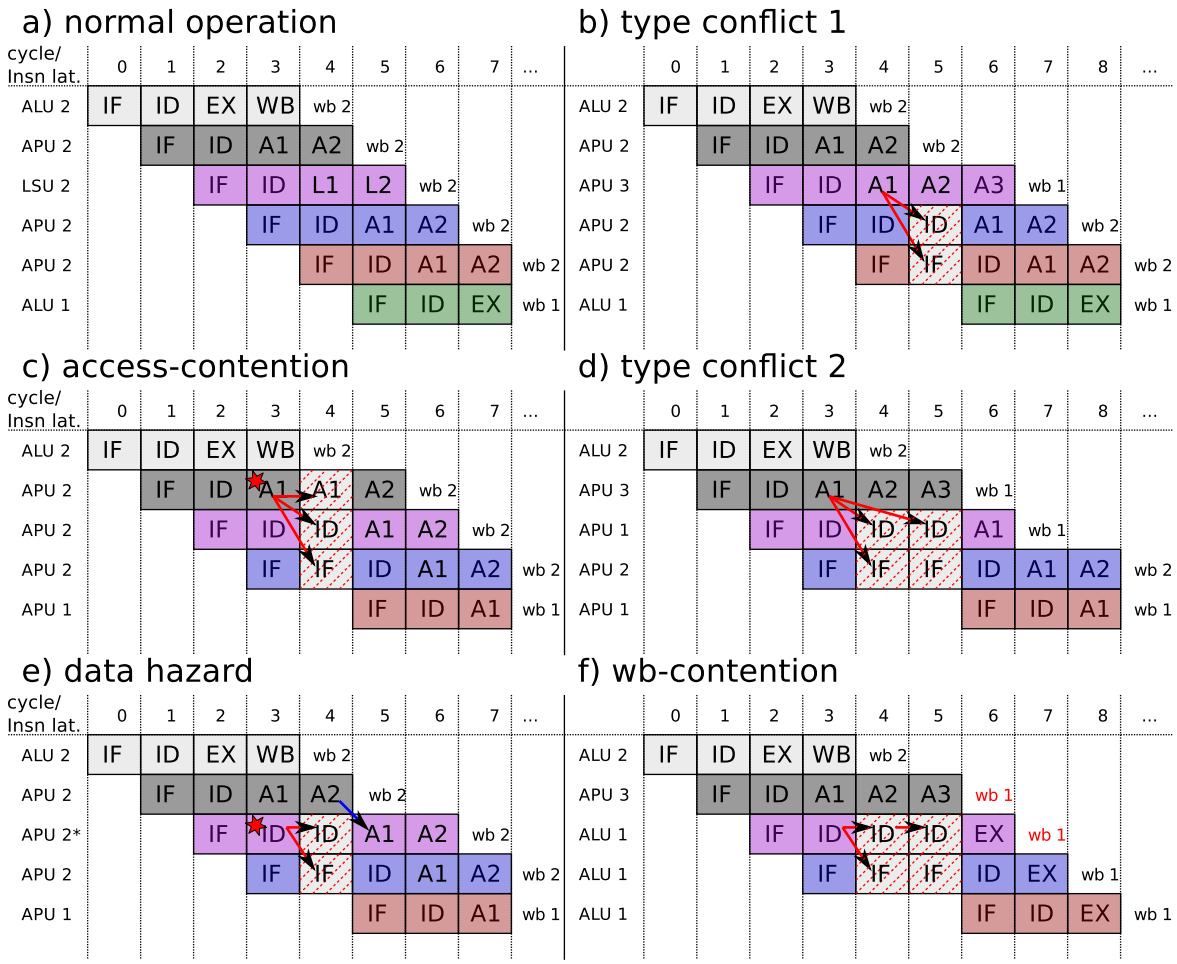


Figure 3.1: Example of six different scenarios when dealing with variable latency APU instructions. a) shows a series of contention free two-cycle APU instructions which execute without stalls whereas the other scenarios lead to stalls due to variable latency, data hazards, access contentions and WB contentions.

the second WB port. Figure 3.1 a) shows how four two-cycle instructions are processed without stalls. Even when interleaved with a load operation, which takes two cycles as well, no stalls occur. Figure 3.1 b) then shows how a three-cycle APU instruction results in one stall cycle because the subsequent two-cycle instruction would finish at the same time, which cannot be handled by the interconnect. This collision is prevented by stalling the pipeline for one cycle. Figure 3.1 c) shows how an APU access-contention results in a stall. The IF-, ID-, and EX-stage will be stalled and the APU-request will be sent until it is granted. Once granted, the next instruction can enter the EX-stage. Figure 3.1 d) shows that the pipeline needs to be stalled for two cycles in case a three-cycle APU instruction is followed by a single-cycle APU instruction. Without stalling in such situations, the instructions would be written back OOO, which would complicate exception handling, debugging, and hazard detection.

Even if the three-cycle APU instruction was followed by another APU instruction which takes three cycles (or even more), the dispatcher stalls the pipeline, because otherwise the dispatcher needs to check for dependencies versus yet another instruction. This would require additional

Table 3.1: Core Interface Signals to interface the APU

Signal Name	Direction	Width	Explanation
		Handshake signals	
apu_master_req_o	output	1	request signal
apu_master_type_o	input	clog2(NTYPE)	type signal to identify shared unit
apu_master_gnt_i	output	1	grant signal from arbiter
		Request channel	
apu_master_operands_o	output	3x32	the three FP- or INT-operands
apu_master_op_o	output	NOP	opcode of the operation
apu_master_flags_o	output	NDSFLAGS	flags for the shared units
		Response channel	
apu_master_valid_o	output	1	valid signal
apu_master_result_i	input	32	result of the shared unit
apu_master_flags_i	input	NUSFLAGS	flags from the shared unit

comparators and make the control logic much more complex. Note, as multi-cycle instructions are typically less frequent, it is highly unlikely that two such instructions are executed in a row. Moreover, it is unlikely that the two instructions do not depend on each other. Hence, this limitation does not result in a significant performance degradation. Due to the increased complexity, and the very rare use-case, the dispatcher does not support OOO APU instructions. Figure 3.1 e) shows how a data hazard is detected when decoding the instruction `APU 2*` which depends on the previous instruction. This is handled the same way as for any other instruction and will stall the pipeline to resolve the hazard. This hazard detection will not only stall the pipeline but also forward the result of the previous instruction to the current one as indicated by the blue arrow. Finally, Figure 3.1 f) shows that if an ALU instruction follows a multi-cycle APU instruction both want to write back the result on the first WB port which will lead to stalls first to prevent OOO WB and second to resolve the WB contention.

3.3 Core Interface

Table 3.1 describes the core interface signals which are used to interface the shared APU. The meaning of the signals is identical to the one described in Section 2.3. Note that the parameters NTYPE, NDSFLAGS, NUSFLAGS, NOP have to match the one of the APU.

3.4 APU Performance Counters

Table 3.2 summarizes the available performance counters which can be used to profile applications which use the shared APU. The four different counters correspond to the events described in Figure 3.1 of Section 3.2. Figure 3.1 b), and d) will trigger an APU_TYPE conflict and Figure 3.1 c) triggers the APU_CONT counter. Figure 3.1 e) increases the APU_DEP counter, and Figure 3.1 f) the APU_WB counter.

As for all other performance counters, on a ASIC target only one counter exists but can be configured to count any event.

Table 3.2: Available Performance Counters to Profile the Shared APU

Performance Counter Name	**Explanation**
APU_CONT	APU access contention: This counter is triggered when more than one core is trying to access the same shared subunit and is not getting the resource. It is a very similar type of contention to TCDM contentions.
APU_TYPE	This counter is triggered when the latency of the current instruction is smaller than the one of a pending APU instruction. The dispatcher stalls in these cases and prevents write back collisions in the interconnect.
APU_DEP	APU inter-instruction dependencies: These stalls occur when the result of an APU instruction is not yet ready but required by the current instruction.
APU_WB	APU write-back contentions. Although, write-back contentions are avoided in the dispatcher, they can still occur with private ALU operations and multiplications which are single-cycle operations. If a multicylce APU operation wants to write back its result and a ALU operation entered the pipeline, it might be that they both want to commit at the same time which will stall the ALU operation for one cycle.

4 Available Functionality

Different shared units have been tested and the most promising candidates are available in the PULP cluster. The shared APU can be configured with a set of parameters which are defined in a single package (`apu_package.sv` located in rtl/ulpcluster).

- **FPU:**
  when set to one, the core is extended with floating-point capabilities.
- **SHARED_FP:**
  when set to one, the FP-signals are connected to the APU rather than to a private FPU. The shared FP-operators are instantiated in the APU.
- **SHARED_DSP_MULT:**
  when set to one, the dot-product extensions are moved to the APU.
- **SHARED_INT_DIV:**
  when set to one, the integer divider is moved to the shared APU.
- **SHARED_INT_MULT:**
  when set to one, the integer multiplier is moved to the shared APU.
- **SHARED_FP_DIVSQRT:**
  defines what kind of FP-DIV/SQRT unit is instantiated.

The SHARED_INT_DIV, and SHARED_INT_MULT parameters are set to zero by default because the amount of saved area due to sharing is small compared to the others. More detailed configuration results are provided for the shared dot-product unit in Section 4.1 and the shared FPU in Section 4.2.

4.1 Shared DSP Units

The first sharing candidates are the dot-product extensions as one of these units alone consumes 6.5 kGE and increases the core area significantly. Dot-product units allow the core to compute up to four multiplications and four accumulations in one cycle which can significantly speed up compute-intensive kernels and at the same time reduce load/store bandwidth. In the most dot-product intense benchmarks (convolutions, matrix multiplications) the utilization of the dot-product unit is between 10% and 30%, making them perfect candidates to be shared in the APU with either one single or two subunits.

Since dot-product instructions are often used in back-to-back fashion, these units have been designed to complete in one cycle. In a shared context, these units will require some extra timing margin for the arbitration between all cores. Since these units were already close to be timing critical, the area-delay behavior of four shared DSP configurations has been evaluated in

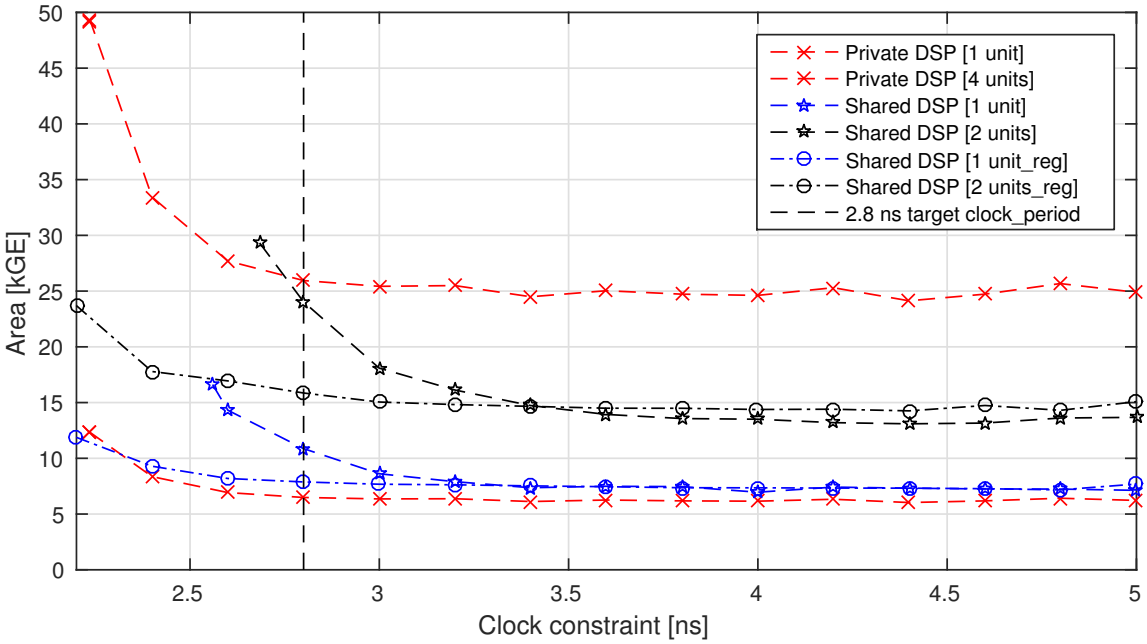


Figure 4.1: Private versus shared dot-product units for a four-core cluster. Shared configurations with one, and two subunits are shown with and without a pipeline stage.

Figure 4.1 to determine if an additional pipeline stage is necessary using a 65nm UMC technology node. The plot shows the area of one and four dot-product units for the private case and one or two dot-product units with one or none pipeline stage for the shared configuration. At a target clock period of 2.8 ns for this technology node, the private dot-product units consume 26 kGE, four times more than a single instance. Sharing only one dot-product unit is the cheapest option, but might lead to many access contentions. Due to the additional area requirement of the $4:1$ arbiter, one shared dot-product unit is always larger than a private dot-product unit, but only 1 kGE. However, sharing a dot-product unit not only increases the dot-product unit by 1 kGE but requires 11 kGE due to the additional timing pressure. Adding a pipeline register to the dot-product unit results in a smaller area but also decreases performance due to the increased latency. Hence, for a single unit, an additional pipeline register can be beneficial but is not yet necessary.

The conclusion changes when more than just one shared unit is required. In this case, the pipelined version is 8 kGE smaller than a single cycle unit because it is much less timing critical. Hence, when sharing more than one unit, it is more area efficient to use a pipelined dot-product unit. Such a pipelined unit increases the delay of each dot-product instruction, which can be hidden in most applications by simply using two independent accumulation registers and interleaving the dot-product instructions. Table 4.1 shows the generated assembly of the inner loop of a 5x5 convolution kernel on byte elements. Five dot-product and sum-of-dot-product instructions are required to perform the 25 multiplications per pixel. In this example, registers `r3-r9` are used to store the weights, and registers `r12-r18` contain the current values of the image patch. The assembly instructions on the left hand side were generated by a convolution with only one accumulation register (`r19`). Five of the seven sum-of-dot-product instructions have dependencies amongst themselves and will stall the core whereas the assembly on the right hand side of Table 4.1 was generated with two distinct and interleaved accumulation registers

Table 4.1: 5x5 convolution with and w/o interleaved accumulations.

Convolution with one accumulator	Convolution with two accumulators

`...`
`start of inner iteration`
`...`
`pv.dotp.b r19, r12, r3`
`pv.sdotp.b r19, r13, r4`
`pv.sdotp.b r19, r14, r5`
`pv.sdotp.b r19, r15, r6`
`pv.sdotp.b r19, r16, r7`
`pv.sdotp.b r19, r17, r8`
`lw r17, -4(r23)`
`pv.sdotp.b r19, r18, r9`
`p.lb r18, r24(r23!)`
`p.addrn r19, r19, zero, 6`
`p.clipu r21, r19, 7`
`...`
`move and shuffle elements`
`...`
	`...`
	`start of inner iteration`
	`...`
	`pv.dotp.b r19, r12, r3`
	`pv.dotp.b r20, r13, r4`
	`pv.sdotp.b r19, r14, r5`
	`pv.sdotp.b r20, r15, r6`
	`pv.sdotp.b r19, r16, r7`
	`pv.sdotp.b r20, r17, r8`
	`lw r17, -4(r23)`
	`pv.sdotp.b r19, r18, r9`
	`p.lb r18, r24(r23!)`
	`p.addrn r19, r19, r20, 6`
	`p.clipu r21, r19, 7`
	`...`
	`move and shuffle elements`
	`...`

(`r19,r20`). This interleaved access allows to resolve any kind of inter-instruction dependencies. In the end, the two accumulation registers have to be summed up which can even be merged with a round and normalize instruction allowing to perform a convolution without additional stalls and without increasing the number of instructions. The only source for stalls when sharing dot-product units are access contentions at the shared unit. With respect to a private implementation, the shared architecture allows to save 10 - 18 kGE depending on the number of required units.

4.1.1 Options

The pipeline depth and the number of dot-products units can be adjusted in the APU. In order to do so, the parameters in the APU cluster package (`apu_cluster_package.sv` located in apu_cluster/sourcecode) can be modified. The parameter `C_DSP_PIPE_REGS` sets the amount of pipeline registers. By default, one pipeline register is inserted which means that dot-product instructions take two cycles in total. If no pipeline stage is desired, this parameter can be set to 0 which will make dot-product instruction single-cycle. By doing this, the latency parameter in the core has to be adjusted accoringly.

The number of units can be controlled with the parameter `NAPUS_DSP_MULT` which is set in the top level file of the APU (`apu_cluster_no_bid_if.sv`). For DSP intensive programs a value of N/2 has found to be optimal, where N is the number of cores. If the unit is not intended to be used very often, it can be set to one.

4.1.2 Instruction Latencies and Usage in C

The sharing is completely transparent which means that no special care is required when programming. The additional latency of the dot-product unit can however lead to more depenency stalls which can be prevented in most kernels by hiding the latency of these operations. The following example in Table 4.1 provides an example of utilizing two independent accumulation registers which allow to process dot-product instructions without any stalls.

Table 4.2: Available FPU configuration options.

Configurations	Parameter names FPU	SHARED_FP	SHARED_FP_DIVSQRT	Est. Area [kGE] per core	per cluster
Private Configurations					
No FP Support	0	0	0		
Units/area per core	none			0	0
Private FPU 1	1	0	0		
Units/area per core	1 FPU (ADD, MUL, CAST), 1 FMA			23	N×23
Private FPU 2	1	0	2		
Units/area per core	1 FPU (ADD, MUL, CAST), 1 FMA, 1 iter. DIV/SQRT			30	N×30
Shared Configurations					
Shared FPU 1	1	1	0		
Units/area per cluster	1 ADD, 1 MUL, 1 CAST, 1 FMA			-	27
Shared FPU 2	1	1	1		
Units/area per cluster	1 ADD, 1 MUL, 1 CAST, 1 FMA, 1 DIV, 1 SQRT			-	56
Shared FPU 3	1	1	2		
Units/area per cluster	1 ADD, 1 MUL, 1 CAST, 1 FMA, 1 iter. DIV/SQRT			-	34
Not supported	all other combinations				

4.2 Shared FPU

Since floating point extensions come at a non-negligible cost, and due to the reason that FPUs are never fully utilized, it makes sense to share the FP-operators in the APU. The fine-grained sharing concept of the APU is perfectly suited to build a scalable FPU for multi-core clusters like PULP. Since the latency of the individual instructions are different, the operations can be grouped in different types which can be accessed concurrently. Further, by sharing the operators, rather than a full FPU allows to better balance the number of units. Finally, by sharing the operators itself, the total number of units can be minimized (usually to one) because the cores can access the different types concurrently.

4.2.1 Options

The current cluster supports six different floating point options which are summarized in Table 4.2. The first option does not support FP operations, and the other five support single-precision FP operations with different number and type of hardware units. The private options come with support for ADD, MUL, CAST through one single unit with an optional support for iterative DIV/SQRT operations. The shared options come with an additional FMA unit, and with different flavours of DIV, SQRT support. Shared FPU 1 does not support DIV/SQRT, Shared FPU 2 supports these operations through a pipelined unit, and Shared FPU 3 with the iterative DIV/SQRT unit.

The number of pipeline registers of the different FP-operators can be controlled with the parameters `C_*_PIPE_REGS` in the package (`apu_cluster_package.sv` located in apu_cluster/sourcecode/).

The number of instantiated FP-operator units can be set in the toplevel file of the APU with the parameters `NAPUS_*`.

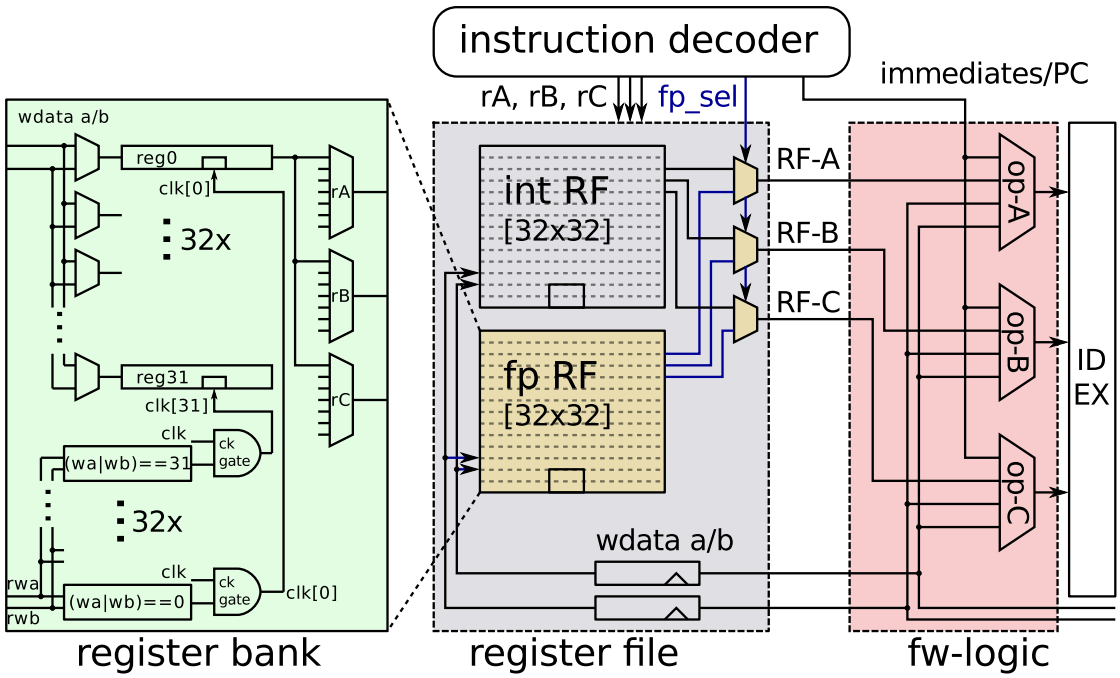


Figure 4.2: Fused integer and floating point register file.

4.2.2 FP Register File

By default FP operands are located in an own register file which is implemented on top of the integer register file as illustrated in Figure 4.2. A merge of the two register files is possible with the restriction that FP register r0 is always 0.0F. The compiler needs to be aware of this merge.

4.2.3 DW Components

For the fine-grained shared FPU, individual operators are prefered over shared units. Since the DW library of Synopsys (or other vendors) offer high quality FP operators, such units are ideally suited to build a shared FPU. On the other hand, single core configurations, where only one operation can be carried out concurrently, prefer a fused data path. DW components come as combinational blocks and are instantiated with a configurable number of pipeline registers either at the input or output of the unit. To achieve an optimal frequency, these registers have to be retimed. Utilizing Synopsys Design Compiler, the following command can be used to retime all FP wrappers:

```
DC> set_optimize_registers -designs [get_designs cluster_fp_wrap*]
```

By sourcing this command before the actual compilation, `compile_ultra` will retime the specified designs in the mapping stage.

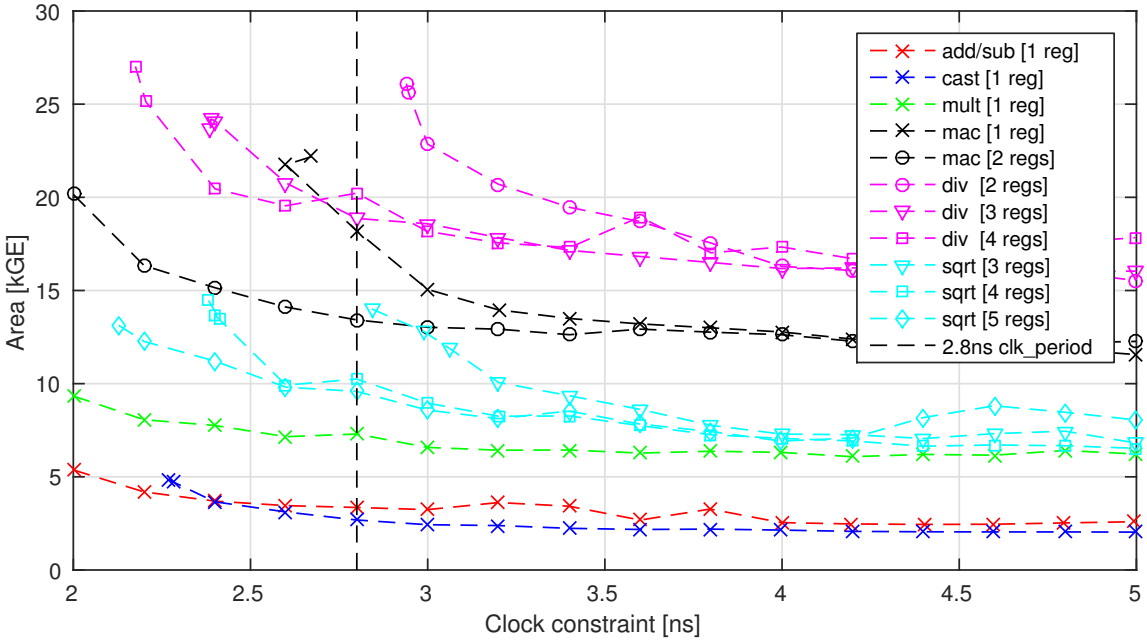


Figure 4.3: Area-delay trade-off of different RISC-V FP components.

Reference Area and Delay Results

Figure 4.3 shows an AT curve of different parametrized DW components when retimed in a 65nm technology and Table 4.3 summarizes the different area consumption and instruction latencies of all FP components at a frequency matching the RI5CY core. The latencies of the individual instructions can be changed by adding/removing pipeline stages to the units as explained earlier.

Table 4.3: Area-delay trade-off of different Synopsys DW FP-operators with a 2.8ns clock period and 0.5ns input/output delay. (UMC 65nm, 1.08V, worst-case conditions)

FP-Operator		**Pipeline registers**	**Latency [Cycles]**	**Area [kGE]**[a] DW	FPU
ADD/SUB	hw	1	2	3.4	
MULT	hw	1	2	7.3	9.6
I2F, F2I	hw	1	2	2.7	
FMA	hw	1	2	18.2	-
		2	3	13.4	
DIV	hw	3	4	18.9	-
		4	5	19.5	
	hw	-	8[b]	-	7.0[c]
	sw	-	65	-	-
SQRT	hw	3	4	14.0	-
		4	5	10.2	
		5	6	9.6	
	hw	-	8[b]	-	7.0[c]
	sw	-	50	-	-

[a] 1 GE = 1.44 µm^2, [b] iterative, blocking unit, [c] Uses the same hardware for DIV and SQRT

5 Future Extensions

Finally, the APU is not limited to the presented extensions, but can be extended with more shared operators. If the following constraints are fullfilled, the ISA extension can be mapped to the APU:

- Not more than three 32b inputs are required.
- The operation does generate an output of maximum 32b.
- The operation only depends on the inputs provided by the cores, and does not rely on other inputs.

5.1 How to move new instructions to the APU

To extend the APU with a new operation, the following steps have to be carried out on cluster level (Global), APU level (APU), and in the core (Core):

Global:

1. Define a new instruction format in the ISA and modify the compiler accordingly.
2. Increase the parameter C_APUTYPES by the number of additional types (see `apu_package.sv`: `C_APUTYPES` has to be equal to the total number of active types)

APU:

3. Assign new and unique type identifiers for the new instructions in the APU top level file (see `apu_cluster_no_bid_if.sv`).
4. Add the desired unit to the APU, add an arbiter, and an interface to connect the arbiter with the shared unit (copy paste from any other shared module).
5. Overwrite the default parameter (`APUTYPE`) of the arbiter (marx) with the previously defined type identifier.

Core:

6. Decode the instruction in the instruction decoder. Assign the signals `apu_en, apu_type_o, apu_op_o, apu_lat_o, apu_flags_o`. Make sure that the type matches the one in the `apu_cluster_no_bid_if.sv` (best practice is to pass it as a parameter).
7. Depending on the latency of the instruction set the `apu_lat_o` signal to `1` (combinational unit), `2` (one pipeline register), `3` (multi-cycle instruction).

Bibliography

[1] M. Gautschi *et al.*, "An Extended Shared Logarithmic Unit for Nonlinear Function Kernel Acceleration in a 65-nm CMOS Multicore Cluster," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 98–112, 2017.