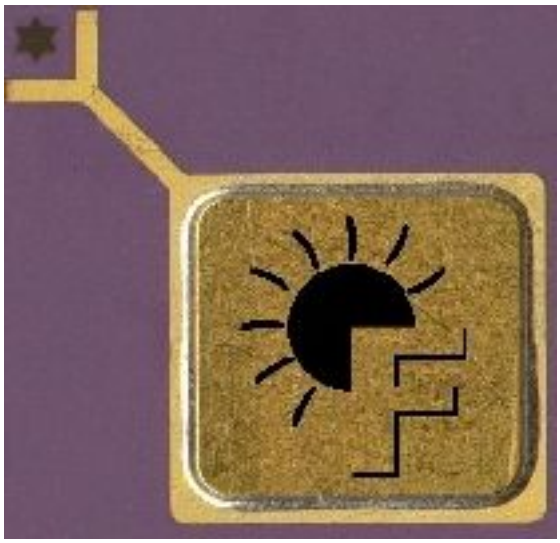


# Introduction

The F-CPU ("Freedom" CPU) is a project to build a purely SIMD superpipelined 64-bit RISC CPU with its sources distributed under the terms of the GNU licence. You can learn much more about it by clicking the logo on the right (or click here ). After reading the F-CPU manual I thought the instruction scheduler could be somewhat improved and made the following proposal. Subsequently many interesting discussions about it took place on the F-CPU mailing list.

## Proposal

Hello.

I read the F-CPU manual. The architecture looks good, OOOC is nice and I particularly liked the SRB. Eventually on seul.org I came across draft 0.2.2, and read about the scheduler (Part IV chapter 3) document p.ps dated 14-Jan-01. This I didn't like so here I propose an alternative solution. It also enables a precise definition of the external view of any execution unit (see recent mail list thread).

---

**Hans Summers**
**http://www.HansSummers.com**

# CONTENTS



# 1. WHAT'S WRONG WITH THE CURRENT SCHEDULER

Here is a copy of the text of the current scheduler.

## 1.1 HIGH COMPLEXITY

The scheduler text does not mention that 2 FIFO's would be required, one for each write port of the Xbar. The design of these FIFO's is also not simple since their slots are allocated at different levels, so data must be writeable by the scheduler to any stage of the FIFO not just the top.

Then we come to the LUT which specifies the precise information about the latency of each instruction. Because we are SIMD the latency can vary if different sized fields are used so the LUT must store latency for all permutations of execution unit and word size. Straight away it starts to look like a large LUT.

## 1.2 BREAKS MODULARITY

Elsewhere in the manual (v0.2 section 3.3) it says "For ease of development and scalability, to name a few reasons, the Execution Units (EUs) are like LEGO bricks that add new computational capabilities to the processor". Ideally therefore each execution unit should be independent, each execution unit should have a standard set of input and output busses and signals, regardless of its function. Incorporation of a particular optional execution unit should require minimal changes elsewhere in the core.

But the current scheduler breaks this modularity: already we have problems with the divide unit and the load/store unit.

The divide unit has (in its simple iterative shift-subtract implementation) a 64-cycle latency and we arrange for this by putting a 6-bit counter on top of the FIFO so it doesn't have to be 64 bits long. At some point as the end of the 64 cycle computation approaches, though, the 6-bit counter will somehow have to reserve a slot in the FIFO for the result of the division unit. How this will be done is not explained, but assuming that someone has a solution in mind or that one is found, it doesn't alter the main point: we have now forced extra complications on the scheduler because of the particular requirements of a certain execution unit.

Similarly with the Load/Store unit: It isn't clear how the current scheduler avoids problems caused by the non-deterministic latency of the out-of-buffer memory access, but obviously the asynchronous Xbar write access must in the case of conflict with existing FIFO slots, stall the execution unit pipelines. More special cases: now the special requirements of one execution unit are not just impacting on the scheduler, but all the other execution units as well, which must now be stallable in the case of asynchronous vs. FIFO conflict.

That's two problems before we even consider other execution units which will also cause problems. For example, what about the integer multiply unit? The mulh instruction stores the 128-bit result of the multiplication of two 64-bit registers in two result registers, r1 and r1 + 1. So it will require a slot in both of the FIFO's, simultaneously writing two output results on the two Xbar write ports. Yet more complication for a specific execution unit - in fact the special cases almost start to look like the rule rather than the exception. This is even before the more complex units have even been considered, e.g. floating point units. The current scheduler will handicap future more complex F-CPU implementations.

Execution Units like LEGO bricks but the bricks will only fit in certain places and putting each

brick in is liable to disturb all the other bricks. I built a lot of LEGO as a youngster and it's unlike any LEGO I remember!

## 1.3 HARD TO MAINTAIN

The F-CPU development is a distributed and collaborative project. Different people design the various execution units, and the other parts of the core e.g. scheduler. The designer of each unit must precisely specify to the scheduler person how many cycles of latency his unit requires, under the various circumstances. The scheduler person must ensure the latency LUT accurately reflects this. Then the designer of a particular execution unit improves his design, and shaves one cycle off the latency. He'd better make sure he talks well and often to the LUT person, or the whole processor isn't going to work.

That's just when the latency of a simple unit changes. What happens when there is a more major improvement? The division unit, which currently uses the simple iterative shift-subtract method gets redesigned and becomes a more efficient pipelined unit. Now we don't just have to update a LUT entry. We have to alter the design of the whole scheduler, removing that special case 6-bit counter which we sat on top of the FIFO.

Wouldn't it be nicer if each execution unit was truly a black-box unit as far as the rest of the core was concerned? And the entire design of a particular unit could be altered without affecting any of the rest of the processor core? This would be conceptually simpler, less likely to go wrong and far, far easier to develop and maintain in an internet environment.

## 1.4 FIXED LATENCY

The current scheduler strictly requires fixed deterministic latency. This is a pity. There are many cases when an execution unit may be able to complete its computation in less than its maximum latency, and the flexibility to do this would improve performance.

For example, consider floating point addition. The lesser of the two operands must first be shifted towards its least significant end by the amount of bits equivalent to the difference in exponents of the two numbers. Then the addition takes place, followed by a possible normalisation (shift) of the result. Several possibilities for optimisation exist here.

First, no addition is necessary at all if the difference in exponents is more than 53. This is because the shift of the lesser operand will make it zero. In this case we just output the larger operand as the result. This is similar to the decimal addition of 640 and 0.03, when I am only considering a 3-digit precision. We don't need to do the addition: the result is 640. The addition

unit can potentially therefore complete in only once cycle, as soon as it recognises this condition.

Second, the post addition normalisation may not even be necessary. This corresponds to the decimal case 83 + 4 = 87. No shifting is required, resulting in reduction of the latency by one cycle.

Other examples would include multiplication where one of the operands is zero (0 * x = 0), or division where the dividend is zero (0 / x = 0). Of course, the decoder could include extra logic to detect these simpler cases, and bypass the execution unit altogether. But that's more execution-unit-specific logic to go in the decoder, and even then I don't see how it could do all the optimisations including things like the floating point addition example above.

This is data dependent latency: wouldn't it be nice if the execution units could themselves recognise where they could optimise, and complete early under such circumstances? Execution units do not have to be designed to optimise like this. Particularly in early days, we may wish to keep things simple. But a better design would allow the black box execution units to be internally redesigned without requiring any change elsewhere in the processor, and invisibly to the instruction set or end user; just functionally equivalent but higher performance. Even if optimisation is not included in the first F-CPU implementation it is wise to design the architecture such that it will be easy to extend in future versions.

## 1.5 BAD PERFORMANCE

The fixed-latency restriction above potentially impacts performance by removing the possibility of data-dependent execution unit optimisation. The current scheduler is OOOC (Out Of Order Completion) but the OOOC is based on the differing latencies of the instructions in the instruction stream, not reordering based on dependencies between the instructions themselves. My proposed alternative solution would perform OOOC based on both the different latency and certain automatic reordering of the completion order when results are required by pending instructions.

# 2 NEW PROPOSAL

## 2.1 INTRODUCTION

My new proposal solves all of the above problems. It should also result in a more precise definition of what constitutes an execution unit. All execution units have the same external structure.

## 2.2 DECODER AND INSTRUCTION ISSUE

The decoder becomes much more simple, it can probably be done in one cycle. I abandon the FIFO completion queues altogether (I'll speak about completion conflicts later). The decoder simply checks to see if the execution unit, source and destination registers applicable to the instruction are available. If they are, the instruction is issued to the appropriate execution unit on the next clock pulse, and the scoreboard "computing" bit of the destination register is set to indicate to subsequent instructions that the register is unavailable as a source register.

This scheme is easily extensible. The instruction decode is so much simpler that it will be easy to construct multiple decoders and issue more than one instruction per cycle in future.

We could even implement OOO (Out Of Order execution) on top of OOOC with a little more thought, if we wanted to. Then we'd need an instruction buffer and instructions which could not yet execute would have to set some scoreboard flag anyway for their destination register, indicating that subsequent dependent instructions could not be performed OOO. Provided dependency was not violated subsequent instructions could be issued OOO. Maybe.

## 2.3 EXECUTION UNIT PIPELINE

Crucial to this design is that each execution unit should have its own destination register pipeline. These replace the original scheduler FIFO's. Each pipelined execution unit has a 6-bit destination register input. The destination value gets loaded into this input when the instruction is issued. That destination value moves along the execution unit pipeline to the output of the execution unit. However many cycles later when the execution unit completes, the destination register number appears at the output along with the result of the computation carried out by that execution unit.

The output register number and result word appear on the execution unit's output, at its Xbar write port. The Xbar then reads the result word and writes it to the specified register. A 7th bit in the pipeline FIFO contains a '1' when the pipeline stage contains a partial computation. The '1' is loaded in when the instruction is issued to the unit, and propagates to the output where it is used as the signal to inform Xbar that the unit is ready to complete.

The latency of the execution unit is not stored elsewhere in the processor core, and it can vary

in a data dependent way, or from one F-CPU implementation to the next.

For example, in an application requiring a lot of integer multiplication, the application designer may wish to use an F-CPU core having an efficient pipelined parallel multiplier. This takes a lot of silicon area, but he may need the performance and decide to replace the standard shift-add iterative multiplier which might be provided in the base F-CPU implementation. Easy: he can just replace the inefficient serial multiplier unit with the parallel one. No questions asked by the rest of the CPU or the software, just better performance from a higher throughput and lower latency.

The only disadvantage to the per-execution-unit destination register pipeline is that it requires higher complexity in the execution units and greater silicon area. However the addition of a 7-bit pipeline is hardly likely to be a significant complication, and given the 64-bit pipeline which will already exist in the execution unit, let alone the execution pipeline stages themselves, proportionally the 7 extra bits won't hurt the area too much. The advantages though are so numerous that I feel this can be tolerated.

An execution unit can even be non-pipelined and use the same mechanism. In this case it can latch the destination register and output it to the destination register output when the execution unit is complete. The simple non-pipelined division unit can still have its 6-bit counter, but it will now be inside the execution unit not the scheduler. At the end of the count, the unit will output the completion bit and the 6-bit destination register number.

## 2.4 XBAR WRITE CONFLICTS

Now I hear you all wailing about what happens when too many execution units complete at exactly the same moment, and the Xbar has only two write ports. The answer is that the Xbar selects only two of the execution units to read from, any other execution units are stalled and wait for a space in the next cycle. A mechanism must therefore exist to stall the pipeline of execution units which have a result ready but the Xbar isn't ready to read it yet. I consider this in more detail a little later on.

How does the Xbar decide which execution register output to read, when more than one is available? Using the same progressive selection "find first" binary tree mechanism described in the SRB chapter (manual v0.2 section 4.3). In this case the inputs to the tree are the 7th bit completion outputs of the execution units.

If there are two write ports in the Xbar, an easy way to select two different waiting execution units is to reverse the order of the execution unit inputs to the find first binary tree. One tree can search from the first execution unit moving forwards, the other effectively moves backwards from the last execution unit.

This also raises the possibility of re-ordering the completion according to priority. Similar to the SRB mechanism, each execution unit can have a "express request" bit output, set to '1' when it urgently needs to complete. I can think of various methods for setting the express request bit.

The express request output of an execution unit could be set when the instruction issue unit is stalled because it is waiting for the execution unit to become available. This could occur often when using long latency non-pipelined units like the divide unit.

A second alternative would be to set the express request bit when the destination register is required by another pending instruction. This though could get more complicated, since the pipeline of an execution stage could contain several destination register computations, how would we detect that one of those is required by the instruction issue unit? In the simple case though, just the completing result could be checked against the dependency of waiting instruction operands.

Thirdly, if the execution unit pipeline is compressible (another idea of mine see later), the express request bit could be set if the penultimate pipeline stage is computing a result. This would indicate that by not selecting the result of that execution unit the Xbar would delay more than one instruction.

Whichever way, it is clear that the Xbar can simply use this method to ensure that there is no conflict on the Xbar's write ports due to multiple simultaneous execution unit completions. At the same time, execution units are free to complete early if they contain the necessary optimisation logic and the operands allow it, and the OOOC does not just reorder instruction execution based on execution unit latency, but also based on dependencies between the instructions. This should all improve performance, modularity and scalability of the design. More execution units can easily be added or replaced with more/less efficient ones depending on the requirements of a particular F-CPU implementation.

# 3 EXECUTION UNIT PIPELINE DESIGNS

## 3.1 INTRODUCTION

In this section I discuss some ideas for the design of the execution unit pipelines, and black box representation of the execution units.

In my examples I don't consider the logic necessary to implement the execution unit's function, just the logic to control the input and output control signals and pipeline. I also don't consider at all the use of the opcode bits that determine which instruction the execution unit will process.

I aim to show that the proposed structure is flexible enough to generically satisfy the requirements of all execution units, and present exciting possibilities for improving the performance of execution units by data dependent optimisation and compressible pipelines. All of which do not alter the external appearance or function of an execution unit, as far as the rest of the processor or software is concerned.

Important note: these ideas are really just "technical cartoons". The actual execution unit input/output signals will have to depend on the precise nature of the Xbar and instruction decoder. The general principle I am proposing is the dropping of the instruction scheduler FIFO and replacement with execution unit FIFO's so that execution units are made more modular and performance is increased.

## 3.2 STANDARD EXECUTION UNIT EXTERNAL VIEW

Under this new proposal, there are no special cases for execution units. All units are treated the same way by the instruction decode and result write. This allows for the precise definition of what inputs and outputs every execution unit must have.

**Inputs:**

--Instruction Load: Signal is set to '1' by the instruction decode/issue to indicate that on the next clock pulse, the execution unit should load supplied operands and start processing

--[0:5] Destination Register [Optional]: 6-bits indicating the destination register of the computation result. This is optional since a Store operation for example does not generate any output

--[0:n] Opcodes [Optional]: Whatever sub-bits of the instruction opcode are required by the execution unit to determine which flavour of operation to perform

--[0:63] Operands [0, 1 or 2]: Input data from the Xbar registers. Some execution units require no operands (e.g. Load), some 1 (e.g. bit shuffler), some 2 (e.g. arithmetic)

--Acknowledge: The Xbar sets this input to '1' to indicate to the execution unit that on the next clock pulse it is going to read the result from the result bus. The execution unit uses this signal to stall its personal pipeline when the Xbar cannot take the result due to a conflict (too many units completing at the same time)

--Clock pulse

**Outputs:**

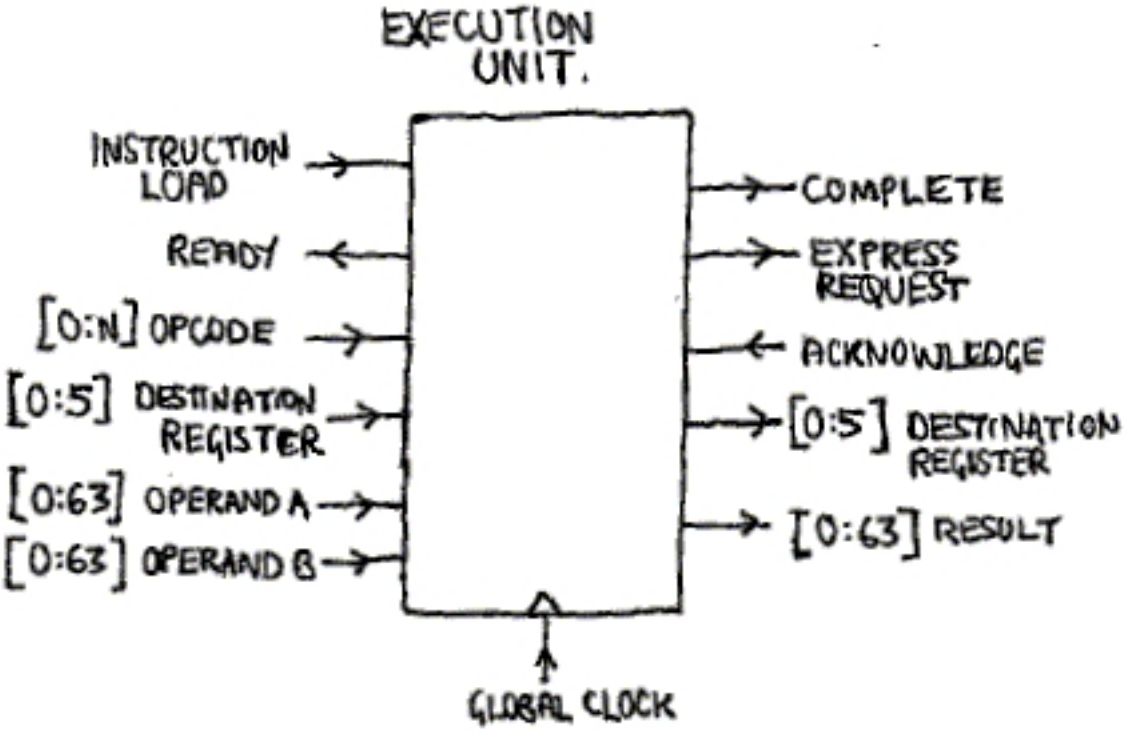
--Complete [0, 1 or 2]: Signal is set to '1' indicating to the Xbar that the execution unit will be ready on the next clock pulse, i.e. its output bus will contain valid result data. For the Store unit which never completes as such (requiring an Xbar write) this signal does not exist. Units having two result busses must supply two independent complete signals

--Express Request [0, 1 or 2]: Signal is set to '1' to indicate to the Xbar that the unit wishes to complete as a matter of urgency. In executions units where this is not implemented, the signal can be hardwired to '0'. Units having two result busses must supply two corresponding express request signals

--[0:5] Destination Register [0, 1 or 2]: The register number of where the execution unit wants the Xbar to write its result data. Optional because a Store for example has no output. The 0, 1 or 2 corresponds one-to-one with the Result set (see below)

--[0:63] Result [0, 1 or 2]. One or more result busses may be presented to the Xbar. They can all be considered independently by the Xbar. Data must be valid at the next clock pulse on the result bus when its corresponding "Complete" signal is '1'

--Ready: A '1' on this output indicates to the instruction decode/issue unit that this execution unit is ready to receive another instruction for processing



**A typical unit having two input busses and one result bus is drawn diagramatically in Fig 1.**

It is important to realise that the execution unit is now a black box. It can determine its own latency, be pipelined or not, and implement its function however it wishes. Externally the function is identical and the rest of the processor core knows nothing about the internal implementation. Execution units are then truly like LEGO bricks.

## 3.3 A PIPELINED EXECUTION UNIT

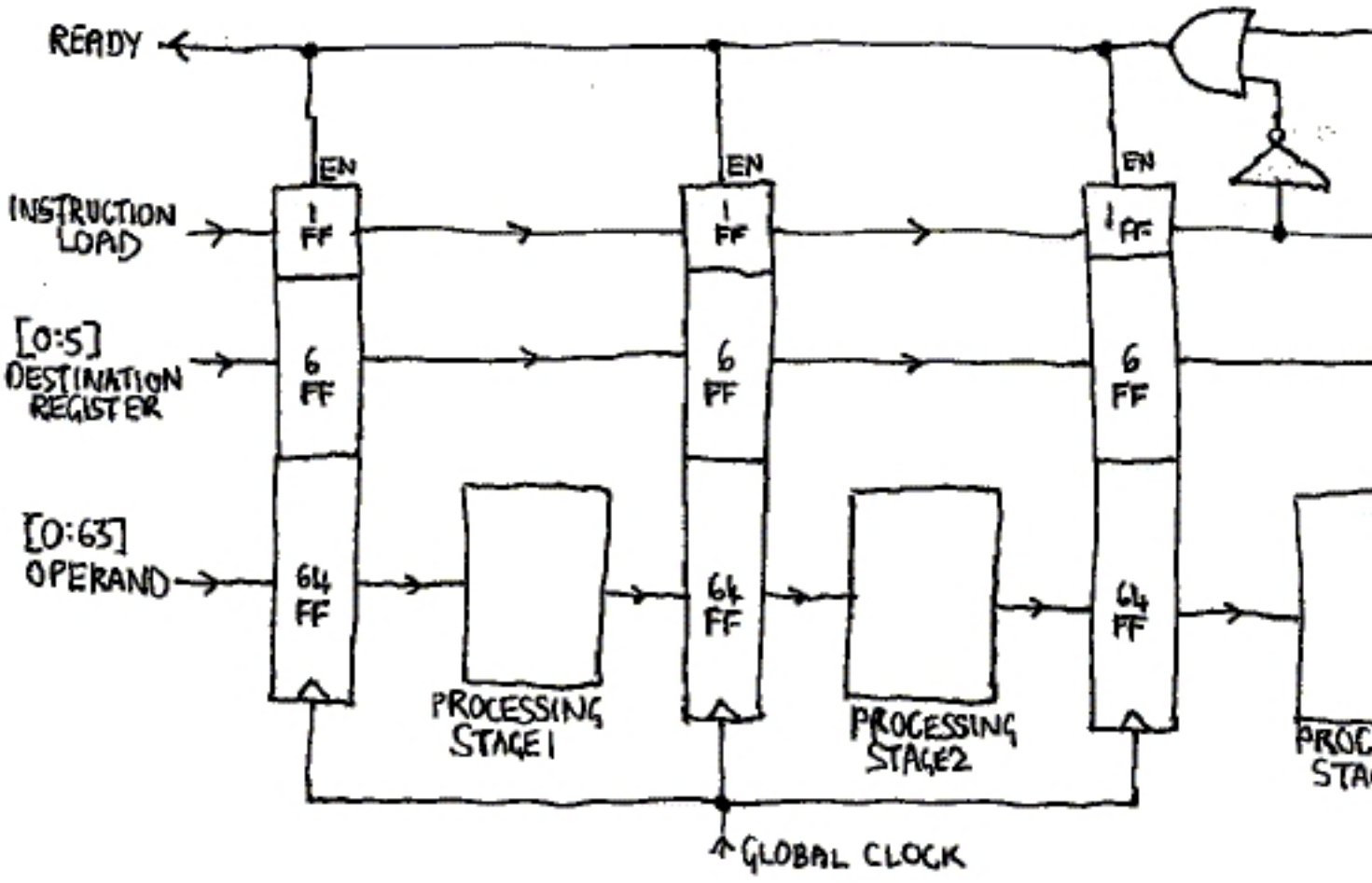


**Fig 2 shows a possible implementation of the internal pipeline logic of an execution unit having one input operand and one result word.**

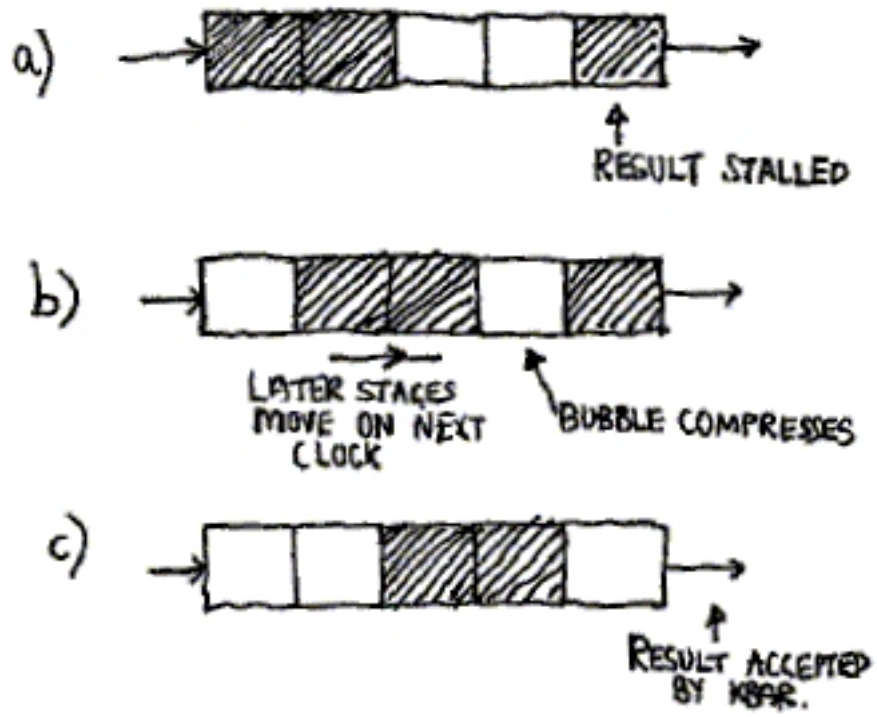
The pipeline latch registers must have a clock enable input to permit easy stalling of the pipeline. The data at the input of the latch is clocked in on the positive-going edge of the global clock only if the clock enable input is high at this time.

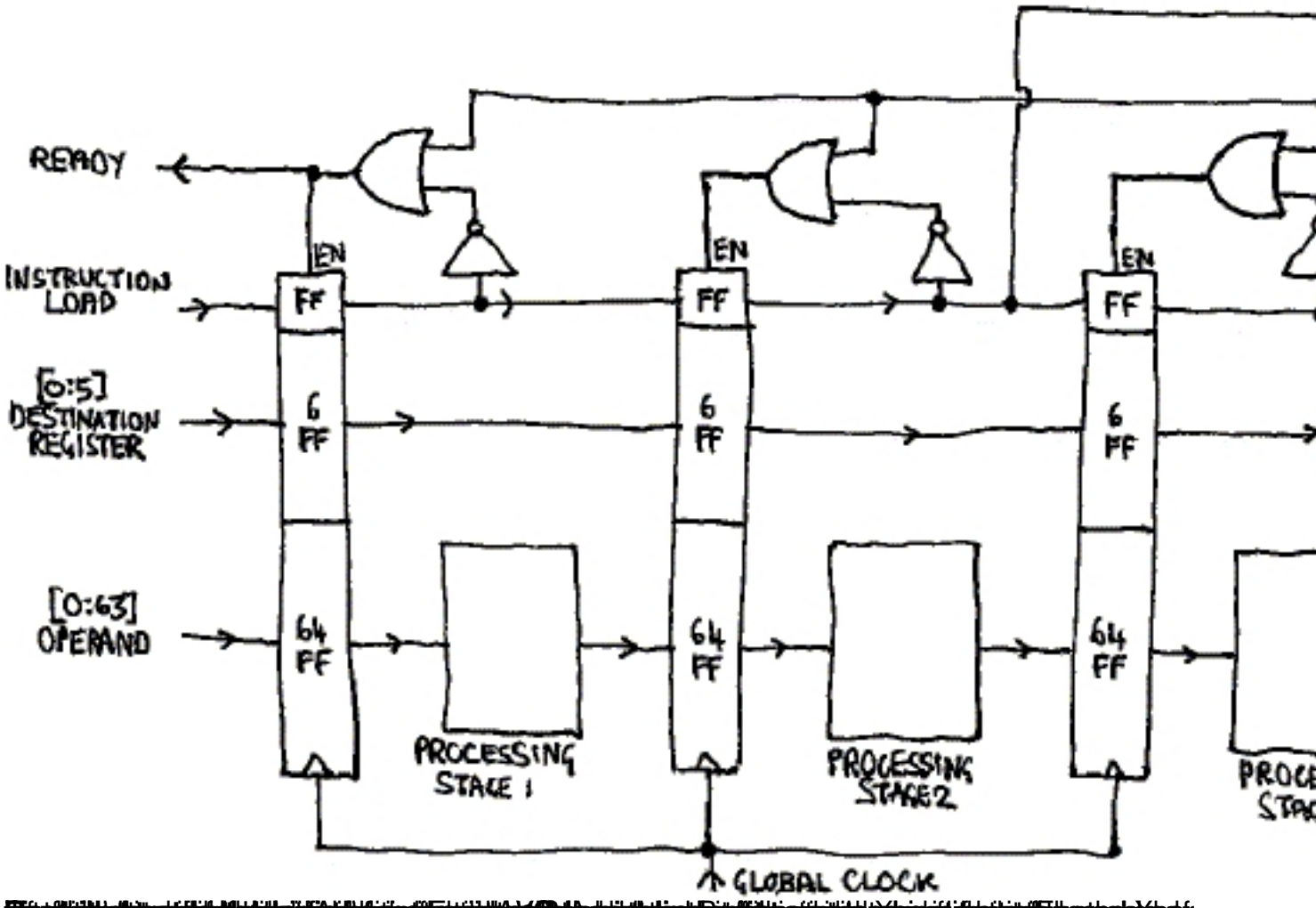
It shows how the 3-stage pipeline is stalled when the Xbar cannot take the result operand. The enable inputs of the pipeline flip flops is '1' when there is either no result pending (Complete = '0') or the Xbar takes the result (Acknowledge = '1'). Therefore when a result is pending and refused by Xbar, the entire execution unit pipeline is stalled. The Ready output comes simply from this enable signal, so that when the pipeline is stalled, the execution unit signals the instruction issue that it cannot accept data on the next clock pulse (Ready = '0').

## 3.4 COMPRESSIBLE PIPELINE

An interesting idea I had is the "compressible pipeline". As the number of execution units is large relative to the number of instructions issued per cycle, it is unlikely that the pipeline of a given execution unit will be fully populated. In this case it is possible to partially stall the pipeline. Imagine the final stage of the pipeline stalls because the Xbar cannot take the result data. If the penultimate stage is empty, there is no need to stall the whole pipeline. Data earlier in the pipeline can still move to the next processing stage, without colliding with the waiting result.

I call it a compressible pipeline because I imagine an industrial pipeline transferring some substance, containing bubbles. If the gas of the bubbles is sufficiently compressible you can imagine that even blocking the output end of the pipe does not stop the movement of the substance further back in the pipe, the bubbles just get squeezed out.


a)
RESULT STALLED
b)
LATER STAGES MOVE ON NEXT CLOCK
BUBBLE COMPRESSES
c)
RESULT ACCEPTED BY KBAR.


READY
INSTRUCTION LOAD
[0:5] DESTINATION REGISTER
[0:63] OPERAND
EN
FF
6 FF
64 FF
PROCESSING STAGE 1
PROCESSING STAGE 2
GLOBAL CLOCK

# 3.5 FORKED PIPELINE

Another interesting idea in an execution unit where the execution pipeline can be forked when early completion is possible. This reduces the latency of the execution unit. For example the previously given floating point addition example. If the unit notices that the addition will simply result in the larger of the two operands, and no addition needs to be performed, and if the final stage of the execution pipeline is empty, the unit can route the operand directly to the final stage and alert the Xbar.

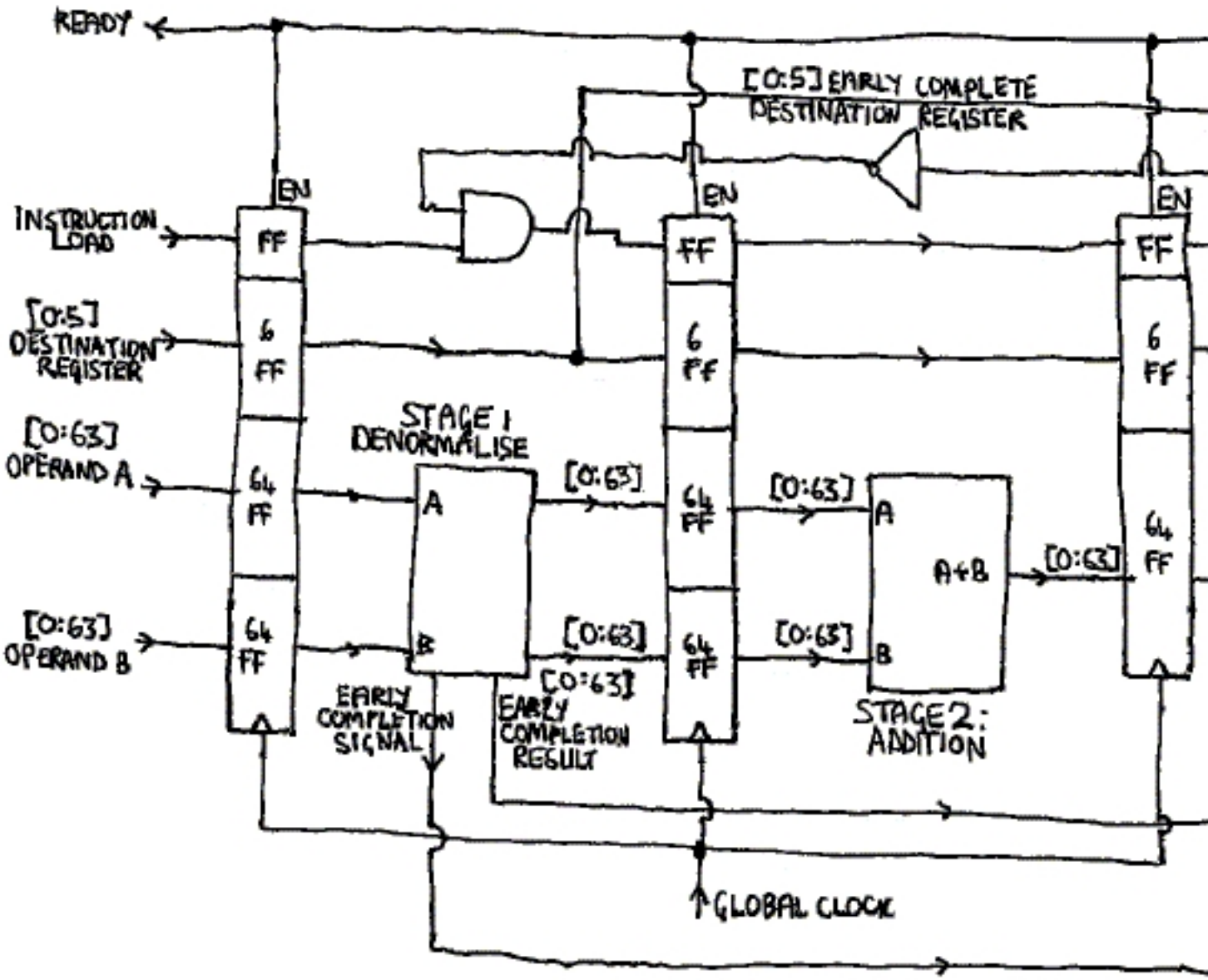


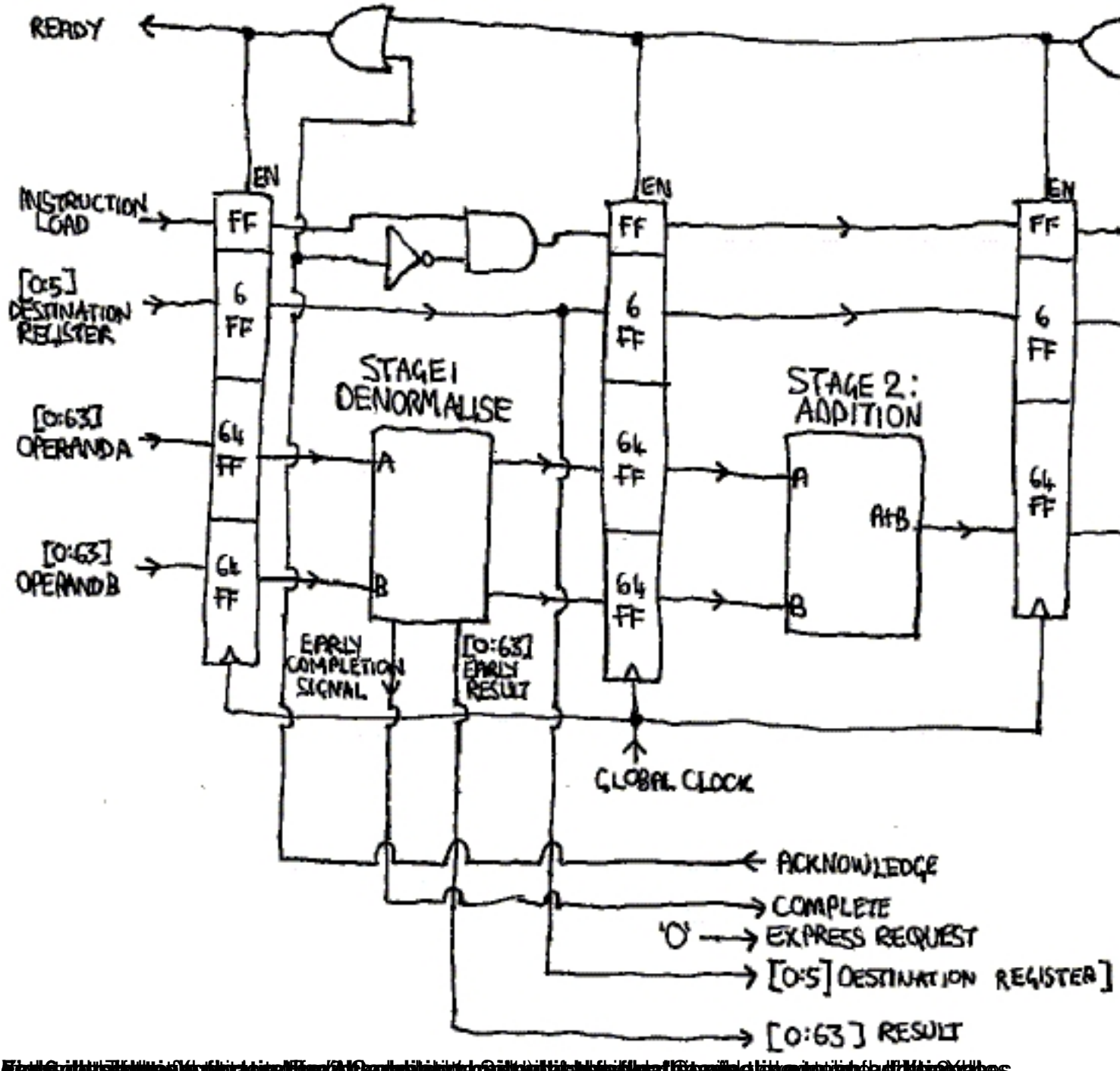
**Fig 5 illustrates the logic of a forked pipeline (simplified) floating point addition unit.**

How does this operate? The first stage produces an early completion result, if it can do so, which is equivalent to Operand A or Operand B. In this instance, it also sets its Early Completion Signal output to '1'. The final output stage is fitted with multiplexers. Provided the final stage is empty, the early completion result is routed directly to the output. The destination register number is also routed from pipeline stage 1 direct to the output stage. The 7th bit (Pipeline Stage Full) at the top of the pipeline register in stage 1 is not passed on to stage 2, which becomes an empty stage. (I pass the data on anyway, but the result from it gets ignored). If early completion is not possible, the execution unit operates as normal.

Note that I am aware that each of my 3 stages may well take more than one stage in the F-CPU, this is just for illustration. The point is that the execution unit can depending on the operand data cut its latency from 3 cycles to 1. Again this is just an optional extra idea, which the architecture is flexible enough to allow should it be needed.

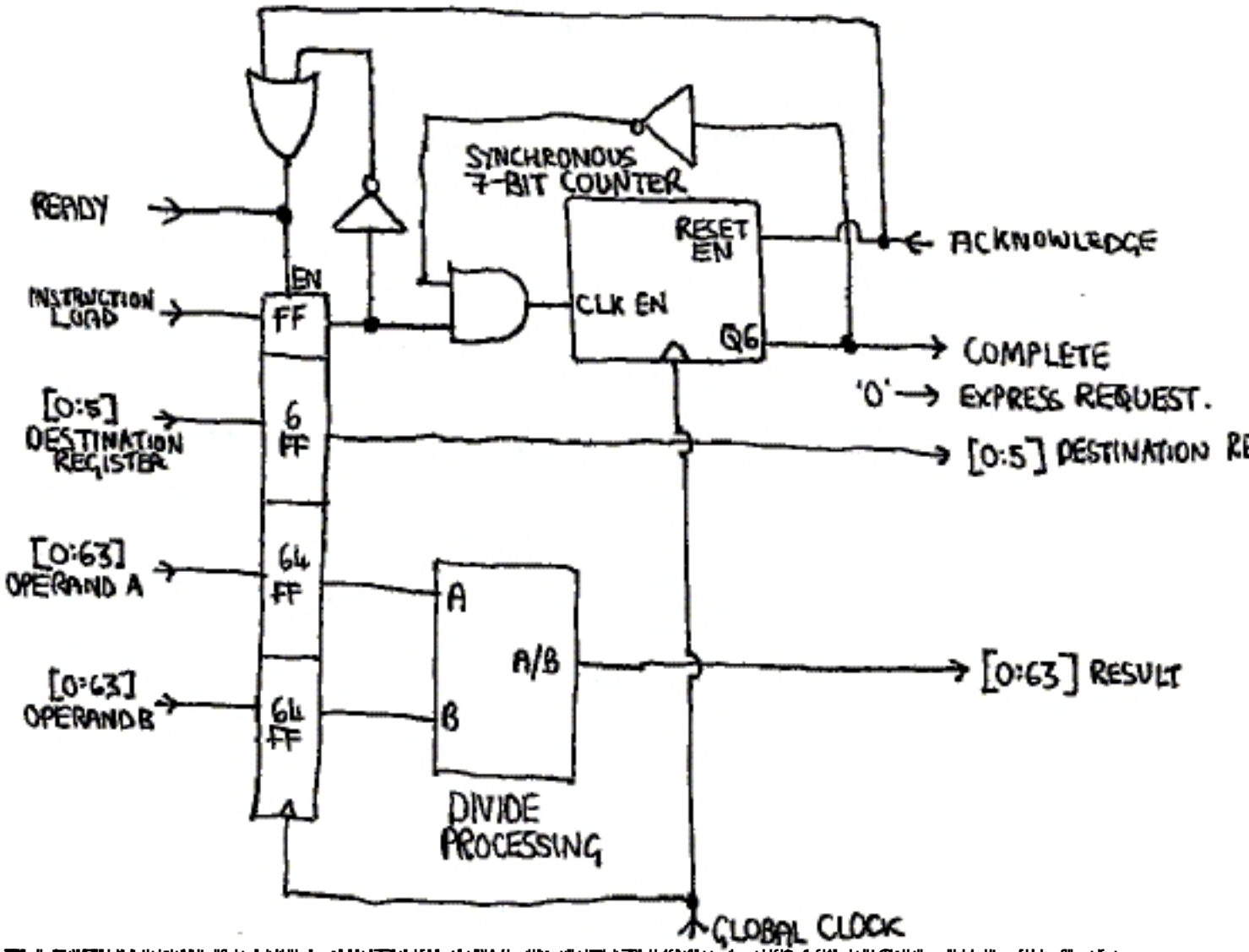
## 3.6 EARLY COMPLETION PORT

Another idea for early completion is to have a separate output port for the early completion result. Such a scheme could be useful if the execution unit was likely to be heavily used and early completion was possible often. In this case the execution unit would have two output ports. They could be activated simultaneously depending on the contents of the execution unit's pipeline. It is the responsibility of the prioritisation activities of the Xbar write mechanism described earlier to handle these outputs.



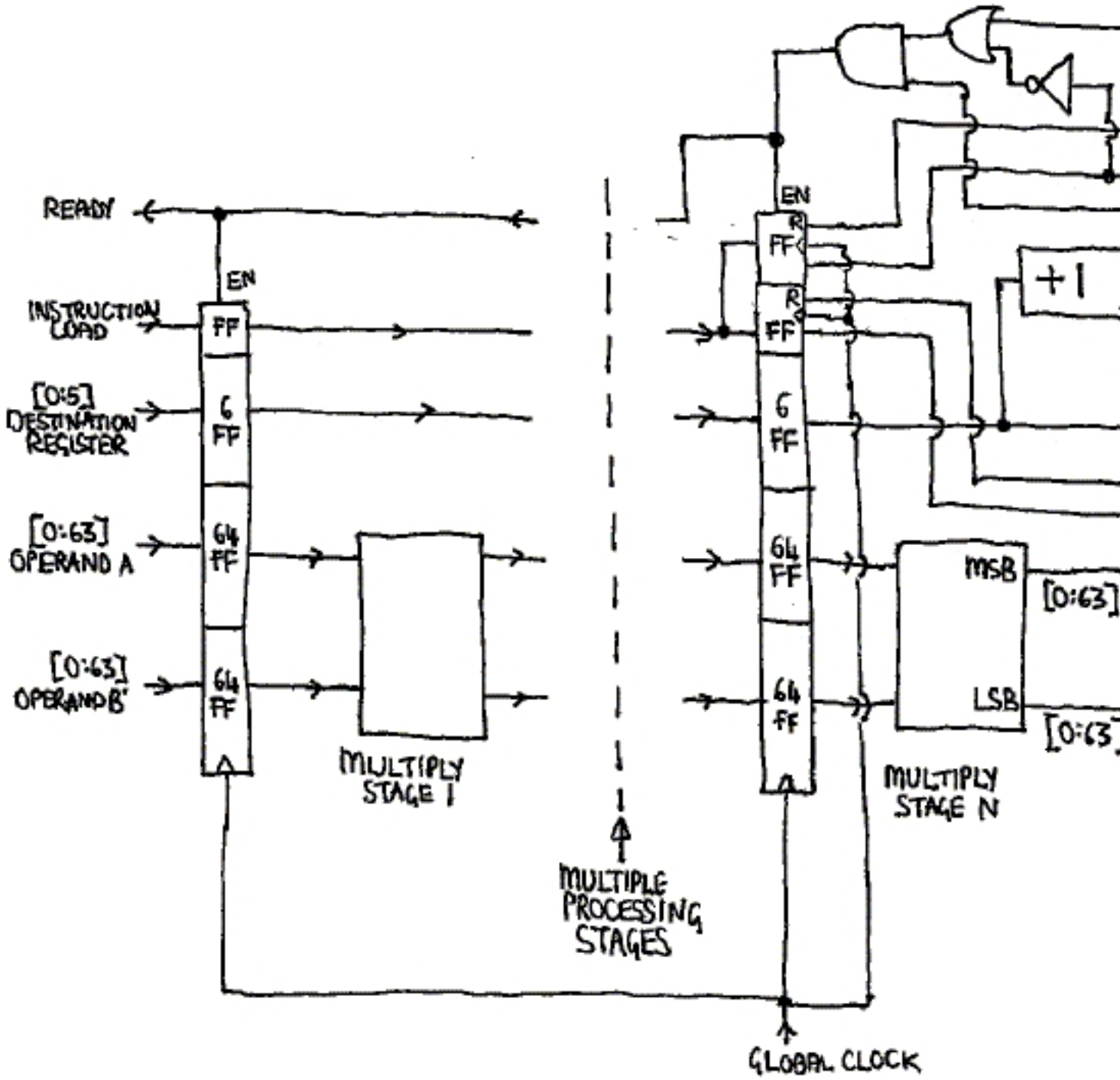

## 3.7 NON-PIPELINED EXECUTION UNIT

Here I consider the design of a non pipelined execution unit (such as the shift-subtract division unit). The destination register is simply latched, and the latency cycles counted by a 6-bit 64 counter. When the count reaches zero, the complete signal is triggered.




## 3.8 DUAL OUTPUT PIPELINED MULTIPLIER UNIT

The integer multiplier generates a 128-bit result (e.g. mulh instruction). This unit can just have two output ports! It is made very simple in this proposed architecture. The destination register output of the high word is just generated by adding one to the supplied destination register. For multiply instructions which don't generate a high word, the second port is simply not used! Again the Xbar will handle prioritisation of reading the results etc. My example design here is only for the mulh case (128-bit result to two registers), in reality one would put extra logic in to decode the opcode bits, since the high word may not be required.




# 5 CONCLUSION

My proposed replacement for the current scheduler unit is conceptually simpler. The decoder/instruction issue unit is made more simple since all it does is check availability and then issue.

It will be easier to develop among a distributed group since the execution units are externally precisely defined. Unit designers can enhance execution units without affecting the rest of the

design or requiring any changes elsewhere (no latency LUT!).

Performance is improved by improving the OOOC (result prioritisation) and permitting interesting optimisation variants of the execution units (non-deterministic latency).

The execution units may truly be considered like LEGO bricks. Different F-CPU implementations for different target users may be built with different execution units, or different variants of execution units depending on the particular performance vs. cost needs of the implementation.

The architecture is easily extensible. It is easy to see how more ports could be added to the Xbar, or the processor made superscalar by issuing more than one instruction per cycle. It is even possible to duplicate execution units if necessary. The complexity of the decoder will remain manageable.

The architecture is innovative. Its principle of operation fits well with the goals of the F-CPU project. The instructions are free. The data is free. The instructions and data flow between the execution units which can adjust their latency at will, and the Xbar dynamically schedules the instruction completion according to priority.