An Instruction Buffer for a Low-Power DSP

M. Lewis, L. Brackenbury
{lewism,lbrackenbury}@cs.man.ac.uk
AMULET Group, Department of Computer Science,
University of Manchester, Oxford Road
Manchester M13 9PL, UK

Abstract

An architecture for a low-power asynchronous DSP has been developed, for the target application of GSM (digital cellphone) chipsets. A key part of this architecture is an instruction buffer which both provides storage for prefetched instructions and performs hardware looping. This requires low latency and a reasonably fast cycle time, but must also be designed for low power. A design is presented based on a word-slice FIFO structure. This avoids the problems of input latency and power consumption associated with linear micropipeline FIFOs, and the structure lends itself relatively easily to the required looping behaviour. The latency, cycle time and power consumption for this design is compared to that of a simple micropipeline FIFO. The cycle time for the instruction buffer is around three times slower than the micropipeline FIFO. However, the instruction buffer shows an energy per operation of between 48-62% of that for the (much less capable) micropipeline structure. The input to output latency with an empty FIFO is less than the micropipeline design by a factor of ten.

1. Introduction

The number of mobile communications devices, particularly cellphones, sold each year is increasing rapidly and a myriad of different products from different manufacturers exist. These devices represent a key application for low-power VLSI design techniques, as battery size and lifetime are among the most important criteria when differentiating between products. Digital cellphones execute complex control and signal processing functions, performing filtering, error correction, speech compression and decompression and, increasingly, additional functions such as voice recognition and image handling. These functions cause the digital components of these systems to consume a significant proportion of the total power.

A common basis for such systems is a microprocessor coupled by an on-chip bus to a digital signal processor. The microprocessor is responsible for the control tasks, and the DSP core handles the complex numerical calculations. An example of part of a current GSM system is the Mitel Semiconductor GEM301 baseband processor [1], which contains an ARM7 microprocessor coupled to an OAK DSP core.

A study of the literature for this product revealed that, within the digital portion, the DSP is responsible for approximately 65% of the total power consumption when engaged in a call using a half-rate speech codec[1]. It could be expected that this proportion of the total power consumption will increase in future generations of GSM chipsets, as the complexity of both the speech codecs and the additional functionality increases. For this reason, we have undertaken a design, as part of the EPSRC/MoD Powerpack project, to investigate the potential of an asynchronous digital signal processor to better meet the performance, electromagnetic interference and power requirements of this application.

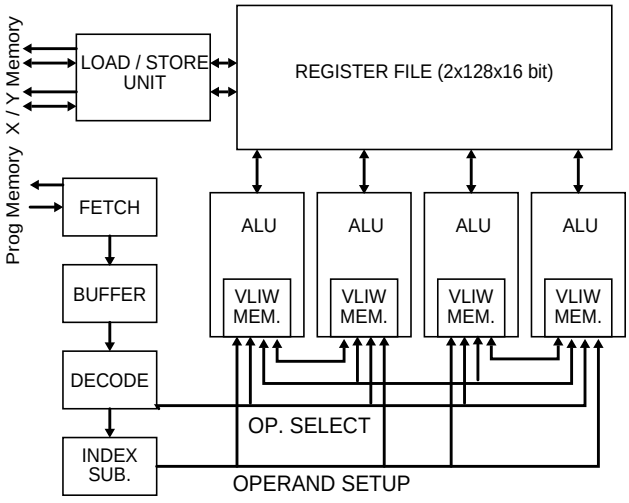

Figure 1: Block Diagram of New DSP Architecture

1. CODEC: COmpresses and DECompresses digital speech data for transmission by radio channel. Half-rate indicates the bandwidth requirement of the encoded data stream.

1.1. A low-power DSP architecture

It has been suggested that the next generation of GSM chipsets will require a throughput of greater than 100MIPs from the DSP, and an initial target of 160MIPs has been set to meet this requirement. An asynchronous design means that any excess speed will simply give longer idle periods with virtually zero power consumption at the end of each processing block.

It has been shown that energy-efficient high performance circuits can be realised by exploiting parallelism [2]. Die area (which is rapidly becoming cheaper) can be traded for increased speed, allowing reduced energy consumption through simpler, slower circuits and reduced supply voltages. The reduced switching rate is also beneficial for EMC, with the natural timing variations of asynchronous design giving further benefits. A parallel structure allows algorithmic transformations to be performed, reducing the switching rate at each unit still further [3],[4]. A structure with four independent functional units has been chosen for this DSP architecture, and a block diagram of the overall system architecture is shown in figure 1.

Memory accesses can be the dominant component of power consumption in data-dominated applications [5], [6]. To reduce the memory bandwidth requirement, a large register file with 2 banks of 128x16 bit words is used in the design. Memory is transferred in bulk to the register file, using RISC-like load and store commands, from where it is accessed by 7-bit index registers which can be updated more quickly and with lower power cost than the wide address registers required in a design where data is fetched from memory via a cache.

Having reduced the power cost associated with data accesses, the next task is to design a method of fetching instructions and dispatching them to the available resources, as the fetching and decoding of instructions makes up a significant proportion of the power consumed by a digital signal processor [7]. DSP activity is often characterized by many repetitions of a few fixed algorithms. This means that it is possible to store very long instruction word encodings for particular operations in advance, in configuration memories internal to the DSP. These very long instructions can then be recalled by a single instruction word. Phillips have used this approach in the R.E.A.L. DSP range [8], and a similar but somewhat more general form of this technique has been developed for the presented DSP architecture. A side effect of the very heavily compressed parallel instructions is that it is possible to write fairly complex DSP kernel routines which require very few instructions to be read from memory.

DSP routines are generally performed using the zero-overhead hardware loops implemented by most DSP architectures. These are simple loops with a fixed number of iterations, as opposed to conditional branches in general purpose microprocessors which introduce possible branch dependencies. In the instruction set for the new DSP, these are performed by the 'DO' instruction. This instructs the DSP to execute the next *m* instructions *n* times, where *m* is a number from 1 to 32, and *n* is between 1 and 65536. DO loops can be exited prematurely by means of the conditional 'BREAK' instruction, whereby the current loop is exited at the end of the pass. Up to 16 DO instructions can be nested, by using an internal stack for the loop status.

This repetition gives a clear opportunity to save energy by means of buffering instructions fetched from memory. On the first pass through the loop, instructions are fetched from memory as normal. In subsequent passes, they are supplied by the instruction buffer. A study of this technique applied to the Hitachi HX24E DSP has shown a power reduction of between 25 and 30% to the total power consumption [9], when a 64-word buffer was added to the architecture. This size of buffer was sufficient to store an 8x8 DCT algorithm, but was insufficient to store more complex algorithms such as FFTs. Due to the compressed instructions in the proposed architecture, it is possible to fit an N-point FFT algorithm entirely within the 32-word limit of the DO instruction. More complex algorithms must either be broken down into subsections, or use the less efficient branch instruction.

The combination of the large register file and loops from within the instruction buffer can massively reduce memory accesses. For example, it is possible to execute a 64 point FFT algorithm with only a single pass of the program and data memories, followed by a single write pass to output the data.

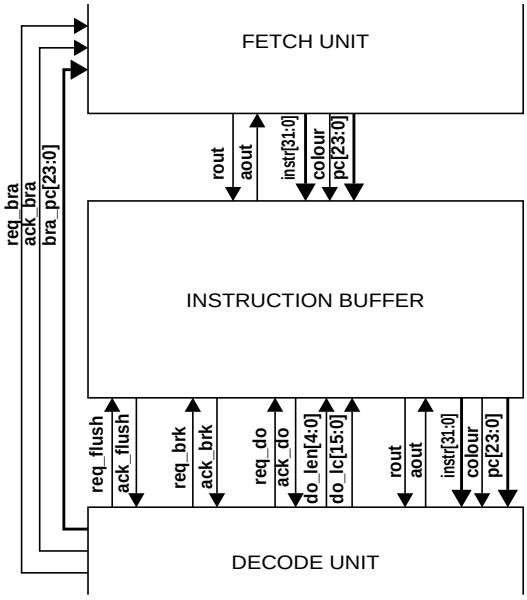

Figure 2: Adjacent pipeline stages and interfaces to instruction buffer

2. Instruction buffer design

2.1. Specification

As is common with asynchronous modules, the first part of the specification is made up by defining the interfaces through which the instruction buffer stage communicates with the neighbouring pipeline stages. A diagram of the connections between the local pipeline stages is shown in figure 2.

The fetch unit forms the previous stage in the pipeline to the instruction buffer. During normal operation, the fetch unit reads instructions from memory and increments the PC value. These are passed to the instruction buffer, bundled by 4-phase request-acknowledge handshake signals [10]. The only variation to this pattern is when a branch instruction is executed, whereby a new value is loaded into the PC. Branches are PC-relative, which means that the PC value associated with each instruction must be passed with it. Any prefetched instructions must be discarded after a branch has been taken, and this is done through a colour tag bit associated with the instructions. This bit is generated within the fetch unit, and its state is inverted when a branch is taken.

Under normal conditions, the instruction buffer simply acts as a 32-entry asynchronous FIFO between the fetch and decode stages. At the output of the instruction buffer, instructions are passed along with their associated colour and PC values to the decode unit, where the appropriate actions are then performed depending on the instruction (or the instruction is discarded, if the colour does not match the current operating colour). In most cases, this forward handshake between the instruction buffer and the decode stage is all that is required, and the first three stages of the pipeline operate in a strictly linear fashion. However, there are three exceptions to this: DO loop setup, BREAK instructions and branches.

For these instructions, it is necessary for the decode unit to communicate back up the pipeline to the instruction buffer, with a reverse handshake on a separate request/acknowledge pair. DO loops are set up by means of the *req_do*/*ack_do* signals and the bundled signals *do_len* (the number of instructions to be repeated) and *do_lc* (the number of repeats to be performed) The BREAK instruction causes the current loop to be exited at the end of the current pass, and this is done through *req_brk*/*ack_brk*. For the case of jumps and branches, it is necessary to exit any loops that are currently in progress, so that the new instruction stream can reach the instruction decode stage. This is done by means of the *req_flush* and *ack_flush* signals.

The basic sequence for each of these reverse handshakes is the same, and is shown in figure 3. At some point after having latched a DO, BREAK or BRANCH instruction and having issued the acknowledge (*aout*), the decode unit sends the appropriate reverse request signal (*req_X*) back to the instruction buffer. The output stage of the instruction buffer will be asynchronously attempting to issue the next forward request (*rout*) during this time. However, this cannot be accepted by the decode unit as it is still occupied by the instruction that set up the reverse request. On receiving the reverse request signal, the instruction buffer performs the appropriate operation. It should be noted that the operation can cause the output of the instruction buffer to change. However, this deviation from the normal data bundling is acceptable as it is under the control of the reverse handshake, and the data is made stable before the reverse acknowledge (*ack_X*) issues from the instruction buffer back to the decode unit. The decode unit can then complete the instruction cycle, and will then accept the forward request from the instruction buffer.

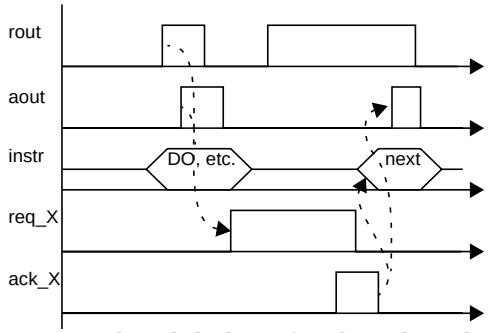

Figure 3: Signal timings for decode unit to instruction buffer communication

2.2. Word-slice FIFO structure

A micropipeline FIFO [10], [11] has the structure shown in figure 4. When a data item arrives at the input, it propagates along the pipeline with each latch closing briefly to store the data until the next stage has acknowledged receipt. This design can have very good throughput, as the cycle time can notionally be reduced to that of a single stage. However, the input to output latency for an empty pipeline is poor as the data needs to pass through every latch. Power efficiency is also poor, as each latch and the associated controller performs an entire cycle when the data passes through it.

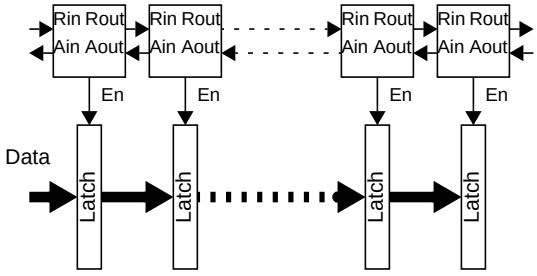

Figure 4: Micropipeline FIFO structure

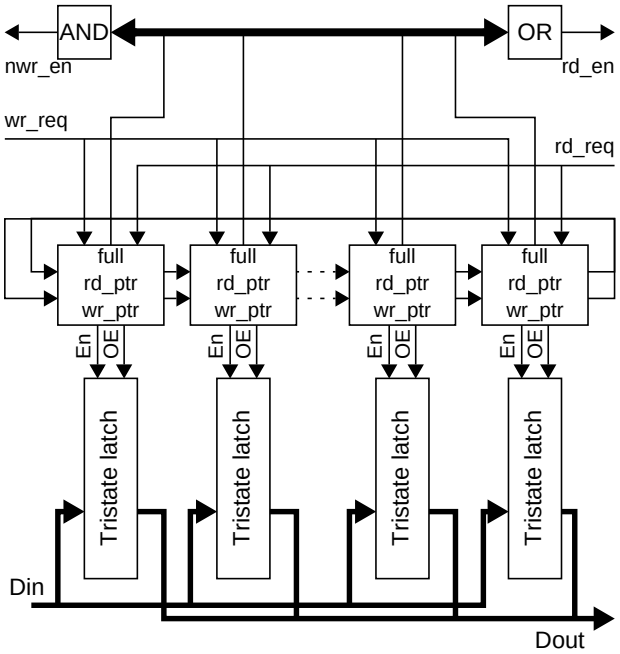

Figure 5: Word-slice FIFO structure

Many alternatives to the linear FIFO structure are possible, which can trade off complexity in the FIFO design against the length of path through which data must travel [12]. However, in order to easily implement the required looping behaviour the *word-slice* structure [13] was chosen. This is a ring-buffer like design, but has distributed rather than central control thus avoiding some of the problems of scalability associated with traditional ring buffer designs [11]. The basic structure is shown in figure 5. The key difference between the word-slice design and the micropipeline design is that the word-slice FIFO has its latch rows in parallel rather than in series, with the outputs multiplexed by means of tri-state buffers. Each row of latches has an associated control element, which controls the write and output enables of the latches and records the current state (full or empty) of the latch. The read and write position is controlled by means of tokens passed around the loop between these latch controllers. Output reads are enabled by an OR of the full indications from all of the latch rows (i.e. a read can be performed as long as there is data to read) and input writes are disabled by ANDing the full indications together. Stability of the AND and OR outputs is ensured by the use of matched delays within the write and read processes. The parallel nature of the structure means that there is only one latch delay between input and output when the FIFO is empty, lowering latency, and the power dissipation associated with the data passing through all of the latches is also eliminated [13].

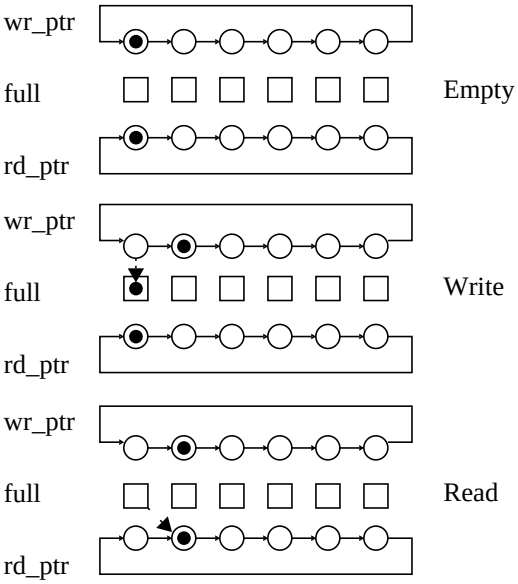

Figure 6: .Standard word-slice FIFO operation

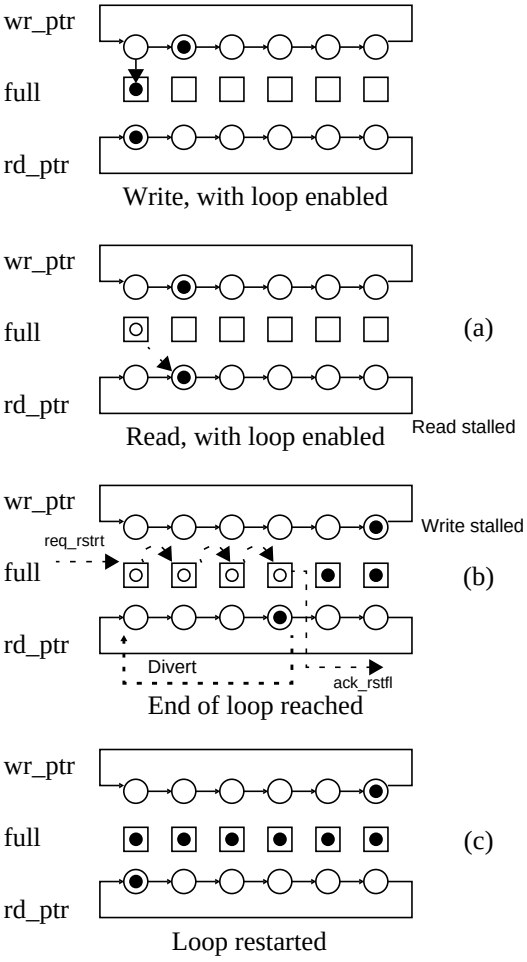

Figure 7: Looping word-slice FIFO operation

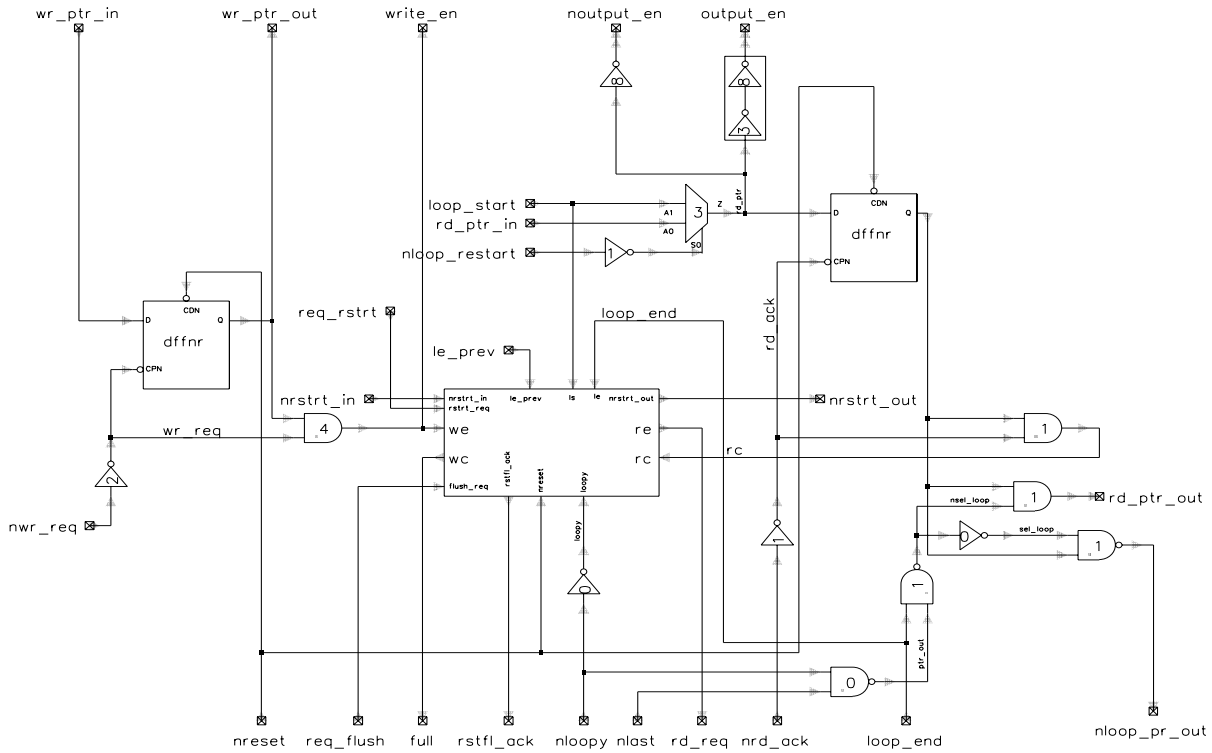

Figure 8: Looping FIFO element

2.3. Looping FIFO design

The operation of a standard word-slice FIFO can most easily be viewed in terms of tokens passing around a ring (figure 6). Each position in the ring buffer has a row of latches which are managed by a latch control unit. These control units have write and read request inputs and an output to indicate whether the stage is full or empty. Two separate overall control units communicate with all of the individual FIFO stages, to interface with input requests and to generate output requests.

When an input handshake occurs, the input handshake controller causes an event on the write input to all of the FIFO controllers. This causes the stage holding the write token to perform a latch write, the 'full' state for that stage to be set, and the write pointer to move one position forward. Write events are blocked when all of the elements hold full states.

The stage that holds the read token makes the latches' tristate outputs active. When any stage indicates that it is full, the output handshake controller produces read requests which, when acknowledged, cause the 'full' state to be reset and the read pointer to be moved on.

When performing a loop, it is necessary to prevent the FIFO stages from being emptied when they are read, so that they can be read repetitively. However, it is necessary for stages that have been read from to *appear* empty to the output controller to stop further output requests being generated if no new data has arrived (an error that could cause the read token to overtake the write token). To avoid this requires a separate 'full' indicator to the input controller and 'read request' signal to the output controller. When performing a loop, read requests from each stage are cleared when the stage is read, without affecting the full indication. This is shown in figure 7a, depicting a full stage with disabled read request by an unshaded dot in the 'full' boxes. When a pass through the loop has completed, a restart signal is issued which causes each of the FIFO stages to appear full again for the next loop. This operation is shown in figure 7b and c. When not in loop mode, or on the final pass through the loop, the output request behaves normally and the stages are cleared entirely when read.

The circuit for the FIFO element that implements this behaviour, shown in figure 8, is composed of three main components: the edge-triggered flip-flops (*dffnr*) on the left and right which store the write and read tokens respectively, and the central handshake controller which manages the full / empty status of the FIFO stage.

2.3.1. Write and read token passing

The write token flip-flops in all of the FIFO stages are connected together to form a circular shift register, with the whole clocked by the *nwr_req* signal from the input controller. The write token enters from the previous stage on

wr_ptr_in, and is accepted when a write request is given to the FIFO on *nwr_req*. Once the element holds the write pointer, a further write request causes the *write_en* signal to go high, which opens the latches in the datapath. When the write request signal is removed, the latches close and capture the new data. The *write_en* signal also indicates to the handshake controller that the stage should become full, which is indicated on the *full* signal to the input controller and the *rd_req* signal to the output controller. As the write is completed, the write token flip-flop is cleared and the token passes to the next FIFO stage through *wr_ptr_out*.

The flip-flops holding the read token also form a shift register, clocked by the *nrd_ack* signal from the output controller. However, to incorporate looping behaviour it is necessary for the token to be passed out of the normal flow to indicate the end of a loop, and for the token to be received again at the start of a loop.

The start and end of a loop is indicated by the *loop_start* and *loop_end* signals to each of the FIFO elements. In normal (non-looping) operation, the read pointer enters through *rd_ptr_in*, and is multiplexed to the read token flip-flop *D* input and the tristate output enable of the latch row. However, when the end of the loop has been reached and the token is to be returned to the start of the loop, *nloop_restart* is driven low and the read token enters whichever stage has *loop_start* high rather than the next stage in sequence.

Similarly, the read token is normally passed out through the *rd_ptr_out* signal. However, when on the last instruction during a loop (*nloopy* or *nlast* low, and *loop_end* high), it is necessary to pass the read pointer out through *nloop_pr_out*, so that the loop count can be updated and the token passed to the correct point to restart the loop.

2.3.2. Handshake controller operation and STG

The handshake controller has four main signals. The *we* input indicates that a write is occurring, which causes *wc* (going to the *full* output) to signal to the input controller that this stage is full and *re* (going to the *rd_req* output) to signal to the output controller that this stage has data to be read. The *rc* input signal indicates that this stage has just been read. The actual effect of *rc* depends on the loop status, as indicated by *loopy*. When not in a loop *rc* clears both the *full* and *rd_req* indications, thereby emptying the stage. However, when in loop mode only the *rd_req* signal is cleared, and the stage remains full. At the end of a pass through the loop *req_rstrt* is asserted, which causes *rd_req* to return high for the next pass through the loop. To allow for nested loops, only the stage at the start of the loop responds to *req_rstrt* (figure 7b), whereupon it signals a restart to be passed from stage to stage around the loop through *nrstrt_in and nrstrt_out*. When the restart signal reaches the end of the loop, it is passed back out on the *rstfl_ack* signal to acknowledge that the restart has completed. The final main signal, *flush_req*, causes any elements that have been read as part of a loop to be emptied, as is required by the flush operation.

The signal transition graph describing the operation of the FIFO handshake control element operation is pictured in figure 9. The STG uses boolean guards on certain transitions, indicated by the question mark. These transitions are enabled only when the boolean condition (after the question mark) is met. For clarity, the placeholders and transition enable conditions for the boolean guard signals in the STG are omitted, as are ineffectual transitions on *flush_req*.

In the reset state, as shown in the figure, a write request causes the *wc* signal to go high, indicating that the stage is full, and the *re* signal to go high, requesting an output read. When a read request occurs at the FIFO stage, the *rc* signal goes high. The response to this then depends on the loop state, as indicated by the *loopy* signal. If a loop is not in progress then the output request *re* is set low, after which the full indication (*wc*) can return to zero and the cycle can restart. If a loop is in progress then the full state is not reset. Instead, the *loop_rd* state variable is set which disables the *re* output. From here, either the loop restarts (*restart_in* is asserted) and the *loop_rd* variable resets, allowing the output request *re* to return to the high state, or *flush_req* is asserted causing *loop_rd* to be reset, and the whole stage to return to the reset (empty) state.

The finished STG specification was successfully synthesised into circuits using the Petrify tool [14], and consists of four C elements and a number of combinational logic gates.

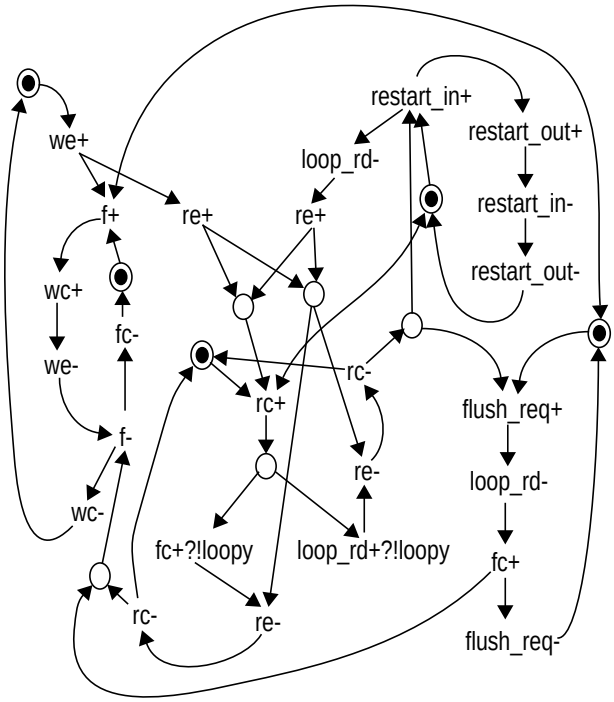

Figure 9: STG for FIFO handshake control

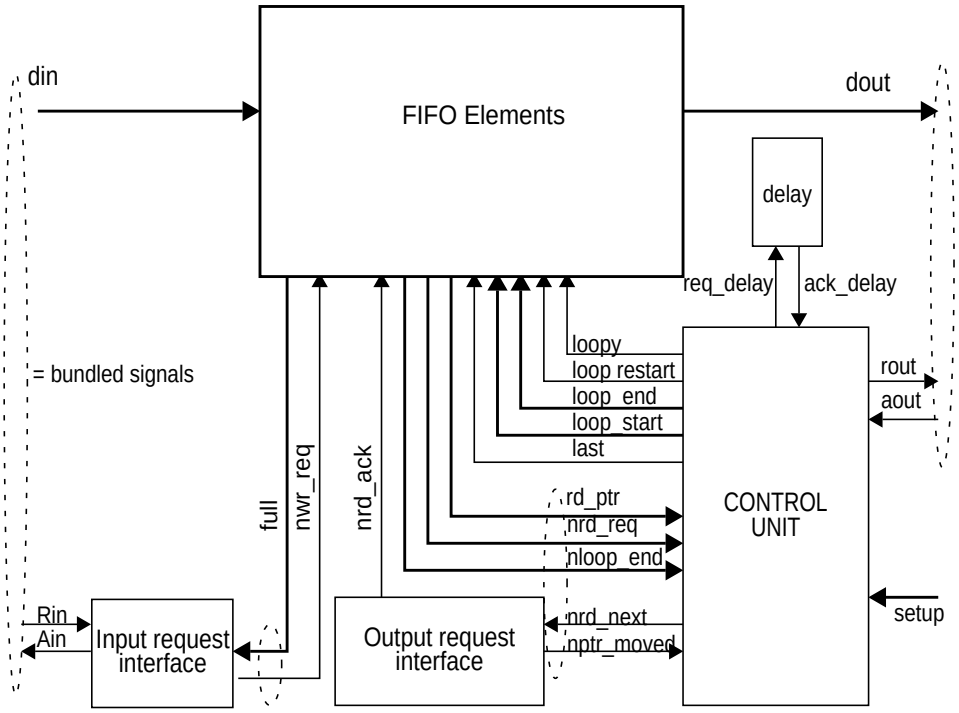

Figure 10: Looping FIFO datapath diagram

2.4. Overall system design

In addition to the FIFO elements already described, the instruction buffer as a whole is made up of 3 other main parts: the input handshake generator that provides a 4-phase input interface, the read handshake generator that provides a 4-phase interface to the FIFO read signal, and the overall control unit. A block diagram of the top level structure, with the interface signals between each stage, is shown in figure 10.

At the input request interface, write requests arrive on *Rin* whereupon the *nwr_req* signal is asserted to perform a write operation and the *Ain* signal is asserted. An internal matched delay is used to allow the write token to move and the full signal from the FIFO to stabilise, after which the input cycle either completes by returning *Ain* low or is stalled if the FIFO is full.

The control unit is the 'brain' of the instruction buffer, and interfaces the FIFO elements to the output, manages loops, and deals with reverse handshakes from the decode stage to set up loops or perform breaks and flushes. By handling both the forward and reverse handshakes at the output, it is possible to ensure that the data remains valid. The control unit is logically divided into the control core, made up of speed-independent logic, and the control datapath which is responsible for storing and updating the current loop status.

The main task of the control unit is to respond to read requests from FIFO elements, by initiating a handshake on *rout/aout*. When the decode stage acknowledges receipt of the data, the output request interface is signalled through *nrd_next* to move the read token to the next position. The timing for the move of the read token and the stabilisation of the signals from the FIFO is also managed by a matched delay, after which *nptr_moved* is asserted.

If the FIFO elements indicate that a loop end has been reached, the control unit updates the loop counter and restarts the loop. On the final pass through the loop, the next outermost loop (if any) is restored.

Once the new token position is known to be correct, a final matched delay is used to mirror the delay from valid tristate FIFO output enables to valid data at the output.

2.5. PC latch scheme

It was mentioned previously that PC relative branch instructions require the associated value of the PC to be passed through the FIFO. This is unfortunate, as branches are comparatively rare instructions in this architecture and the requirement to store the PC initially seems to require an additional 24x32=768 latches which is a great waste of power and area. Fortunately, the sequential nature of the PC values means that this overhead can be greatly reduced. The instruction buffer contains a maximum of 32 sequential PC values, which means that, unless a carry out is generated from bit 4 of the PC, the upper 19 bits of the PC remain constant. A carry out will be reflected by a change in bit 5.

This behaviour is altered slightly when branches are considered: in this case, the PC can change to a random value. However, when a branch is taken the instruction colour tag is changed so that the decode stage can discard prefetched instructions in the branch shadow before any other instructions can occur. It is therefore possible to store only the lower 6 bits of the PC in the FIFO, and to use 4 sets of latches to store the upper 18 bits. One of the 4 latches is enabled for writes, based on the value of bit 5 of the input PC and the current input colour. Similarly, only one of the 4 latches is enabled for output by bit 5 of the output PC and the output colour. This saves a total of 504 latch elements.

2.6. Control datapath design

The control datapath, as shown diagrammatically in figure 11, is internal to the control unit and maintains the current loop status. It is driven by the control core which handles all of the complex interactions between the signals from the FIFO datapath and the reverse requests from the decode stage. The control datapath consists of a row of latches that holds the current state (loop start and end position, *first*, *last*, and *loopy* status, and the current loop counter). When a DO loop is set up, the current position of the read pointer from the FIFO datapath (encoded into 5-bit binary) is added to the requested number of instructions to make up the loop. The current read pointer and the result of the calculation are used to set up the new loop start and end positions. Before the new loop status is loaded, the old status (if any) is pushed onto the 16-entry stack. When the loop is exited, the stacked data is reloaded and the stack is popped, thereby allowing nested loops.

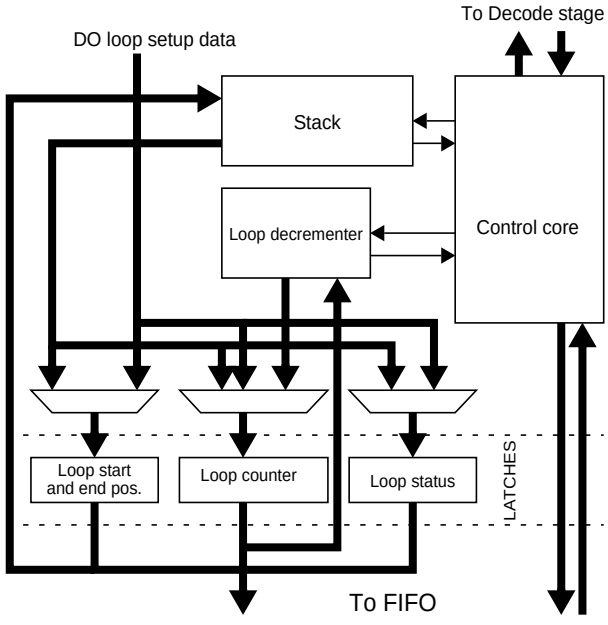

Figure 11: Top-level diagram of control datapath

On each iteration of the loop, the control core requests that the loop counter unit decrement the value of the loop counter (although the loop counter is actually stored in inverted form and incremented). In parallel with this, the result is checked to see whether it will be zero, which indicates the last iteration of the loop.

The loop counter uses a simple ripple-carry incrementer, but has data-dependent matched delays to achieve good average-case performance. The delay is selected by looking at adjacent 4-bit groups of the value being incremented. Should any of the bits be zero, then it is known that the carry cannot propagate beyond that point and the delay corresponding to the worst case propagation up to that group is selected.

2.7. Control core circuit design

The operation of the control core is dependent on various FIFO state signals (such as the loop status, the read pointer positions, and the loop count value). The FIFO datapath and the control datapath are designed so that the control core can treat these signals as being bundled with handshake interfaces. Having the stable periods for these signals defined makes it possible to produce signal transition graphs for the correct sequences of operations, depending on the state of these signals, which can then be synthesised into speed-independent circuits using Petrify.

Signal transition graphs with large amounts of choice proved to be very confusing to design, due to the number of enabling arcs between state signals and different transitions. In order to simplify the design process, the control was broken up into six sections each of which is concerned only with a limited number of the state signals. This meant that understandable (and synthesizable!) signal transition graphs can be produced for each section, and their outputs combined where necessary with a certain amount of glue logic. The structure into which the control core was split is shown in figure 12.

The heart of the control is the *handshake controller* circuit, responsible for issuing read requests to the FIFO and producing output requests when the data has stabilised. After each output is produced, a request is passed to the *loop control* circuit. If the FIFO is not at the end position of a loop then the loop controller immediately issues an acknowledge to allow the next output cycle to proceed. If the FIFO is at the end of a loop, then the loop counter must be decremented and the loop restarted before the acknowledge is returned. If the loop counter reaches zero, the '*last*' flag is set, allowing the FIFO to exit the loop after the last pass, whereupon all of the loop variables (including the state of the '*last*' and '*loopy*' flags) are popped from the stack to restore any previously nested loops.

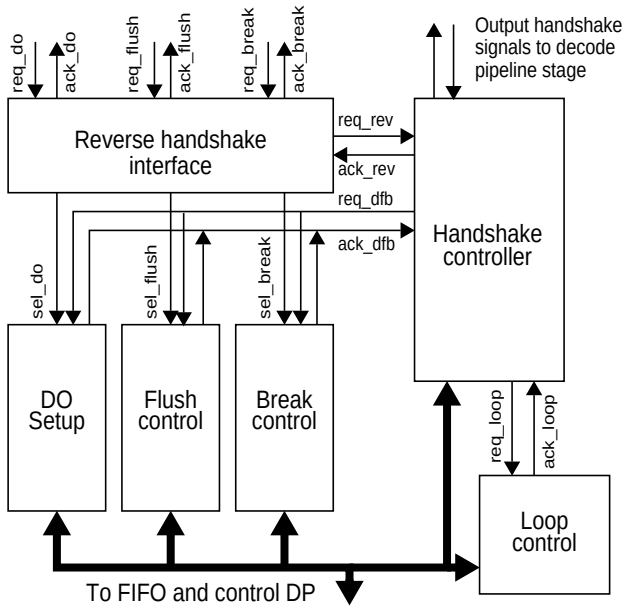

Figure 12: Control core structure

To simplify the design of the main handshake control circuit (which is the critical path for the common case instructions), requests for DO setups, breaks and flushes are dealt with by the *reverse handshake interface* which means that the handshake control circuit need only deal with a single interface. When a reverse request arrives (*req_do*, *req_flush* or *req_break*), this sets up a state signal (*sel_do*, *sel_flush* or *sel_break*) and issues the *req_rev* signal to the handshake controller. When the handshake controller is in the appropriate phase of its cycle to respond, it issues a common request to all of the DO setup, break and flush units (*req_dfb*) and the unit selected by the *sel_xxxx* signal performs the appropriate operation, indicating completion on *ack_dfb*. This allows the handshake controller to issue the *ack_rev* signal and the reverse handshake to complete.

3. Tests performed

All testing of the instruction buffer was performed on netlists extracted from schematics, as the DSP is still under construction and has not yet moved into the layout phase. The initial verification of the design, during and after the design of the circuits, was done with the instruction buffer in situ, as part of the DSP pipeline shown in figure 2, executing test programs under the *TimeMill* simulator. A selection of loops and nested loops were performed successfully. In addition, the loop counter unit was tested with a separate C simulation model, to set up and measure the delays for each level of carry propagation both within the loop increment circuit itself and for the incrementer cycle time including the time to latch the new value.

Once the functionality had been verified, a new testbed was designed in which the instruction buffer could be tested in isolation. This consists of a C simulation model that feeds random instructions, using sequential PC values with random branches, to the input of the buffer at a selectable rate. The output from the buffer is then captured and compared with the value that should be present, and the latency from the input to the output of the buffer is measured.

As a baseline with which to compare the instruction buffer, a 32-element 4-phase micropipeline FIFO [10] was also designed (the 4-phase asynchronous interface making it easily interchangeable with the instruction buffer). The same tests were performed with the micropipeline design.

Two sets of tests were performed, using the *PowerMill* simulator to compare power and performance figures. The first set of tests fed 500 random values through each buffer at the maximum rate at which it would accept them. The second set of tests fed the same 500 values through each buffer at intervals of 20ns, which was significantly slower than the cycle time for both circuits. This models the case of the memory being slower than the stage into which the FIFO is feeding, and measures the latency from input to output. In both cases, current consumption was measured for each design.

4. Results

4.1. Loop counter performance

The delay figures for the loop count incrementer are shown in Table 1. The delays are shown for the four different possible groups of carry chain length.

Assuming a count from 0 to 65535, the mean delay can be calculated from the formula:

$$\bar{d} = \frac{15 \times (4096 \times d_3 + 256 \times d_7 + 16 \times d_{11} + 16/15 \times d_{15})}{65536}$$

The results that have been obtained give a mean delay of $\bar{d} = 2.31ns$, which is close to the minimum delay. A count over the full range should be near to the worst case for the mean delay: shorter counts will miss out some of the longer carry propagate chains.

Table 1: Incrementer delays

Max. number of carry stages	Inc. delay (input to output request) / ns	Loop counter cycle time / ns
3	0.66	2.25
7	1.41	3.13
11	2.48	4.33
15	3.12	5.04

4.2. Throughput results

Table 2: Maximum throughput and minimum latency

	Cycle time	Throughput	Latency
Instruction buffer	6.0ns	167MHz	2.7ns
Micropipeline	2.0ns	488MHz	26ns

4.3. Energy consumption results

Table 3: Energy consumption per cycle

	Average energy per input cycle	
Rate	Maximum	50MHz
Instruction buffer	0.32nJ	0.48nJ
Micropipeline	0.67nJ	0.77nJ

5. Discussion

The comparison between the instruction buffer and the micropipeline FIFO shows the instruction buffer to have a throughput that is less than that for the micropipeline design by a factor of three (although the micropipeline design does not have the additional circuitry required to perform looping). However, the micropipeline FIFO exhibits a latency that is a factor of ten greater than the instruction buffer. The cycle time results are acceptable, being much less than the 25ns cycle time dictated by the DSP application, even when added to the worst-case loop counter increment time. The low latency will ensure that instructions pass from memory to the decode unit as quickly as possible. Naturally, these figures will be degraded somewhat when interconnect delays and capacitances are taken into account but should still easily meet the specification requirements.

It was observed during testing that the bulk of the cycle time was required for the tri-state outputs of the latches to drive the broad output array. In a design that requires greater throughput it would be possible to split the outputs into two or more sections, with a controller for each section that moves a read pointer at a rate reduced by factors of two for each subdivision. This would allow the design to be scaled to an arbitrary degree, with the number of gate delays from input to output increasing only by the logarithm of the number of stages.

Compared to the micropipeline FIFO, the word-slice instruction buffer exhibited reduced energy per data value transferred in both the test cases, giving an energy per input of 48-62% of the energy for the micropipeline design. The fact that the instruction buffer outperforms the much simpler micropipeline FIFO is evidence that this was a good choice of circuit structure for low power. It also illustrates one of the key benefits of asynchronous design: while the instruction buffer has much more circuitry than the micropipeline FIFO, much of the circuitry in the instruction buffer is inactive during normal operation, and being idle consumes virtually no extra power. The arguments for splitting the tristate outputs into sections could also be applied to power consumption, by reducing the switched capacitance at the output. However, this would probably only be of benefit for larger sizes of buffer. Later results with back-annotated capacitances from the final layout should better answer this question.

6. Conclusions

The design of an instruction buffer and its role in a low-power DSP architecture has been presented. The word-slice FIFO structure has been augmented to produce complex looping behaviour, and yet offers a significantly lower latency and power consumption than the much less complex micropipeline FIFO.

More detailed results gained from simulations of extracted layout and use of the circuit in its place in the DSP pipeline should provide some useful lessons about the design's strengths and weaknesses. The nature of the design is such that it could be scaled up or down relatively easily with only slight changes to the overall structure should that be required at a late stage in the design process.

7. Acknowledgements

This work formed part of the EPSRC/MoD Powerpack project, grant number GR/L27930. The authors wish to express their gratitude for this support.

8. References

[1] *GEM301 GSM Baseband Processor Preliminary Information*, Mitel Semiconductors 1997

[2] A.P. Chandrakasan, R.W. Brodersen, "Minimizing Power Consumption in Digital CMOS Circuits", *Proc. IEEE* vol. 83 no. 4, April 1995

[3] T. Arslan, A.T. Erdogan, D.H. Horrocks, "Low Power Design for DSP: Methodologies and Techniques", *Microelectronics Journal*, vol. 27, no. 8, pp. 731-744, Nov. 1996

[4] A.T. Erdogan, T. Arslan, "Low Power Multiplication Scheme for FIR Filter Implementation on Single Multiplier CMOS DSP Processors", *Electronics Letters*, vol. 32, no. 21, pp 1959-1960, 1996

[5] F. Catthoor, “Energy-Delay Efficient Data Storage and Transfer Architectures and Methodologies: Current Solutions and Remaining Problems”, *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology*, vol. 21, no. 3, pp 219-231, 1999

[6] K. Danckaert, K. Masselos, F. Catthoor, H.J. DeMan, C. Goutis, “Strategy for Power-Efficient Design of Parallel Systems”, *IEEE Transactions on VLSI Systems*, vol. 7, no. 2, pp 258-265, 1999

[7] H. Kojima, D. Gorny, K. Nitta, K. Sasaki, “Power analysis of a programmable DSP for architecture / program optimization”, in *Tech. Dig. IEEE Symp. Low Power Electron.*, pp. 26-27, Oct. 1995

[8] P. Kievits, E. Lambers, C. Moerman, R. Woudsma, “R.E.A.L. DSP Technology for Telecom Baseband Processing”, *Proc. 9th Intl. Conf. On Signal Processing Applications and Technology*, Miller Freeman, Inc., 1998

[9] R. S. Bajwa, M. Hiraki, H. Kojima, D.J. Gorny, K. Nitta, A. Shridhar, K. Seki, K. Sasaki, “Instruction buffering to reduce power in processors for signal processing”, *IEEE Transactions on VLSI Systems*, Vol. 5 No. 4, pp. 417-423, Dec. 1997

[10] S.B. Furber, P. Day, “Four-Phase Micropipeline Latch Control Circuits”, *IEEE Transactions on VLSI Systems*, vol. 4 no. 2, pp. 247-253, June 1996

[11] I.E. Sutherland, “Micropipelines”, C*ommunications of the ACM*, vol. 32, no. 6, pp 720-738, June 1989

[12] E. Brunvand, “Low-latency self-timed flow-through FIFOs”, in *16th Conference on Advanced Research in VLSI,* Chapel Hill, NC, IEEE Computer Society Press, pp. 76-90, 1995

[13] K. Yi, “The Design of a Self-Timed Low Power FIFO using a Word-Slice Structure”, M.Phil. thesis, Department of Computer Science, University of Manchester 1998

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, “Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers”, *IEICE Transactions on Information Systems*, vol. E80-D, no. 3, pp. 315-325, March 1997