nnARM Architecture Specification

version 1.18

Writen By ShengYu Shen
From NUDT
2001.8.1
Proof Reading By Clayton John
2001.8.2

NOTE

This document describes the architecture of the nnARM processor core. Every major release of the nnARM has only one such document. Any enhancements of nnARM v1.18 will not be included in this document. For documentation of newer changes and features, please refer to newer versions of the documentation and the comments in the source code.

Release Log

V1.00	**2001/4/11**
V1.10	**2001/6/1**
V1.11	**2001/6/10**
V1.18	**2001/8/1**

Not aN ARM

1.Introduction

The nnARM project is a development project started on March 24, 2001. The purpose of this project is to develop a synthesizable high performance embedded processor core that can execute the ARM v4T instruction set.

At this time, the v1.18 of this soft core has been completed. It contains the following components:

1. a behavioral description of a memory controller.
2. Behavioral descriptions of an instruction cache controller and a data cache controller.
3. a RTL synthesizable instruction prefetch buffer.
4. a RTL synthesizable instruction fetch component.
5. a RTL synthesizable decoder for Thumb instructions
6. a RTL synthesizable decoder for ARM.
7. a RTL synthesizable full function ALU that can support all ALU operations used in the ARM instruction set.
8. a RTL synthesizable pipeline "mem" stage that can perform load and store operations.
9. a RTL synthesizable register file.
10. a RTL synthesizable interrupt processing module

The whole ARM v4T instruction set is summarized below. All Thumb instructions are translated into the corresponding ARM instructions. So, whether a particular Thumb instruction can run depends on whether the corresponding ARM instruction can run. The already supported instructions are shown in green, unsupported instructions in red:

1 Data process/PSR transfer
2 Multiple/Multiple accumulate
3 Multiple long
4 Single data swap
5 Branch and exchange
6 Half word data transfer
7 Single data transfer
8 Block data transfer
9 Branch/branch with link
10 Coprocessor instructions
11 Software interrupt

Note: The Tomasulo structures have been removed because I can not manage to deal with the complexity of designing them. Also, I think that the logic and interconnect resources on a FPGA will be too limited to justify using such complex structures.

Not aN ARM

This documentation is organized in the following way:

Section 1: Introduction.
Section 2: Describes the overall architecture of nnARM.
Section 3: Describes the storage hierarchy of nnARM.
Sections 4-7: Describe the 4 pipeline stages.

2. Overall architecture

2.1 Structure Introduction

A top-level block diagram of the processor is shown in figure2.1 below.

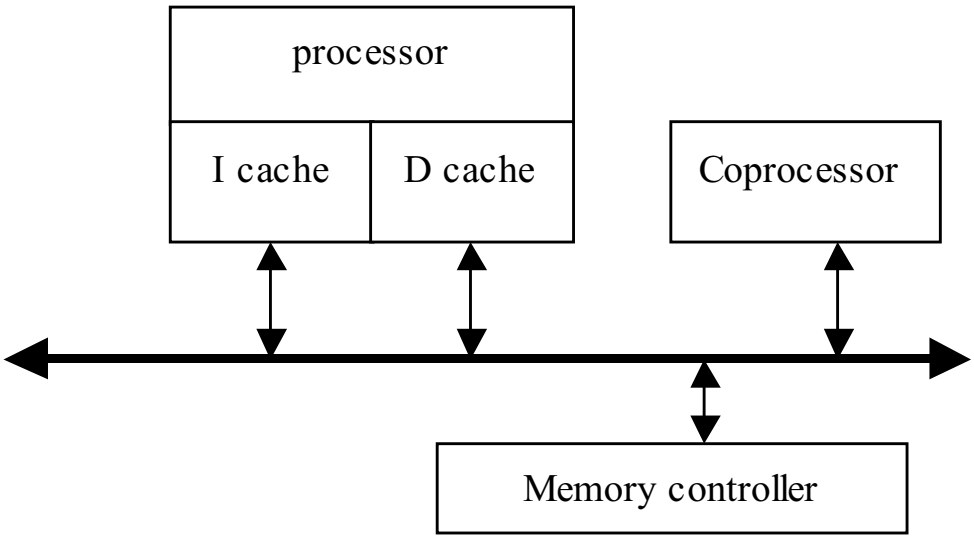

figure 2.1

The processor has separate data and instruction caches. Both caches are coded in behavioral level Verilog code descriptions. The detail of both caches will be given in the next section.

The memory controller is also coded as a behavioral description. It is not part of the current design focus, so a behavioral description is sufficient for it.

The coprocessor can accept requests from the main processor through the memory bus. It can distinguish between memory requests and coprocessor requests. Although the coprocessor interface has been shown in this diagram, and some consideration has been given to it, it is not implemented in the current design. It will probably be added later.

The pipeline is very similar to that of DLX or MIPS, It contains only 4 stages: IF ID ALU and MEM. There is no WB stage for register write back, I have merged it into the MEM stage to simplify the pipeline design.

The first stage is IF (Instruction Fetch). It is describe in IF.v file.It fetches one instruction from the prefetch buffer every cycle. This is a so called "1 issue pipeline." 2 issue or 4 issue operation is possible, but it

would seriously increase the complexity of the overall architecture. Also, a report from ARM states that a two issue pipeline will achieve only a 20% performance improvement. I think that because the ARM instruction set is more like a CISC than a common RISC, its code is relatively dense, so that one instruction word can do more work (on average) than a comparable RISC instruction word. So a single issue pipeline is should be sufficient to realize high enough levels of performance.

The second pipeline stage is the ID (Instruction Decode) stage. It is described in Decoder_ARM.v. In this stage the decoder translates a single ARM instruction into microinstructions and sends the microinstructions into the rest of the pipeline structure.

Since the last release, a Thumb decoder has been added. It translates Thumb instructions into ARM instructions, and feeds them to the ARM instruction decoder. A Thumb flag in CPSR indicates the current state. The Thumb decoder is described in the files: Thumb_2_Arm.v and ThumbDecoderWarper.v

After instruction decode, the microinstructions go to the ALU (Arithmetic Logic Unit) stage, where the various types of computations are actually performed, including: and, eor, sub, rsb, add, adc, sbc, rsc, tst, teq, cmp, cmn, orr, mov, bic, and mvn. Also, at the same time a booth multipler is coupled with the ALU to perform MUL and MLA operations. The ALU is described in ALUShell.v.

After the ALU stage has performed the desired operation, it passes the result and the micro operation to the MEM (MEMory) pipeline stage. In the MEM stage, load/store operations perform accesses to or from memory. Also, at the end of this stage all results for the register file are written back. The MEM stage is described in the file mem.v.

I think I must say a few words about the use of pipeline forwarding. If instruction n uses register Rn as its destination, and the following instruction n+1 uses Rn as its source operand, when n completes its ALU stage, and n+1 is ready to begin its ALU stage, the results of n have not yet been written into Rn! Since instruction n+1 requires the the results of instruction n to compute its result, then a forwarding scheme must be performed to pass the n result directly to instruction n+1.

The following figure illustrates this more clearly.

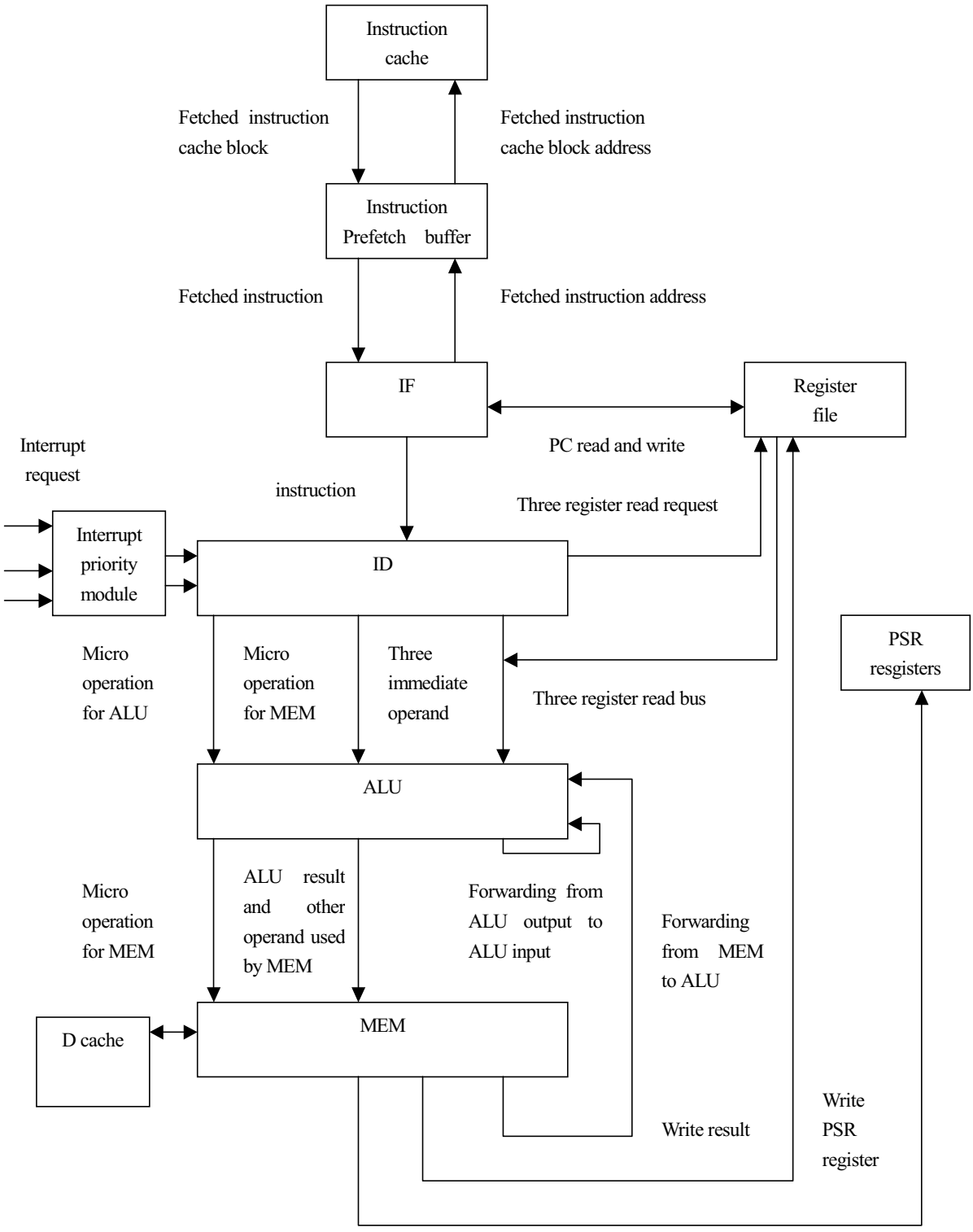
Instruction cache
Fetched instruction cache block
Fetched instruction cache block address
Instruction Prefetch buffer
Fetched instruction
Fetched instruction address
IF
Register file
Interrupt request
PC read and write
instruction
Three register read request
Interrupt priority module
ID
PSR resgisters
Micro operation for ALU
Micro operation for MEM
Three immediate operand
Three register read bus
ALU
Micro operation for MEM
ALU result and other operand used by MEM
Forwarding from ALU output to ALU input
Forwarding from MEM to ALU
D cache
MEM
Write result
Write PSR register

figure 2.2

2.2 Memory Endian Type ("Endianness")

This processor current only supports little endian access. A WORD is stored so that the least significant byte is at the lower address. The address of the WORD is the address of its least significant byte. The memory organization is shown below.

Big endian access is not supported now, and I do not have any plans to support it.

The nnARM uses BYTE addressing, so to move to the next word you would add 4 to the address.

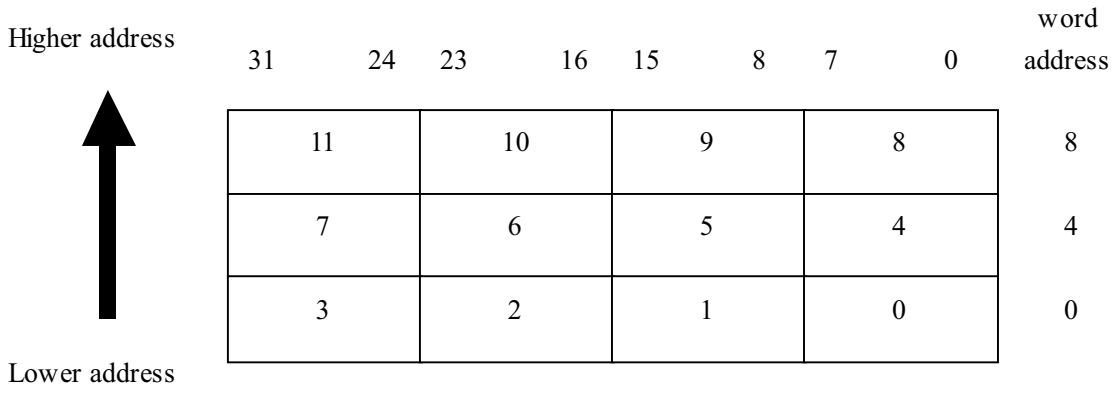

figure 2.3

2.3 Address Bus Width

The nnARM is a brand new design, so it does not need to maintain backward compatibility. Therefore, only a 32 bit wide address bus is supported; the 26 bit address bus mode that was used in ARM7 processors is not supported, and I have no plans to support it.

2.4 Processor Modes

The nnARM processor supports six operation modes:

1) **User mode: The normal program execution mode**
2) **FIQ mode: To support a data transfer or channel process**
3) **IRQ mode: General purpose interrupt handling**
4) **Supervisor mode: Protected mode for the OS**
5) **Abort mode: Memory fetch failure**
6) **Undefined mode: An undefined instruction executed**

Mode changes are controlled by software or external interrupts or exceptions. Most user programs execute in user mode. The other modes (which are used to handle interrupts or exceptions) are called privileged modes.

2.5 General Register file

The register file now contains 31 general purpose registers. The active set of registers is determined by the processor mode.

At any time, 16 registers can be accessed by software. They are R0

to R15. R15 is the program counter, other registers can all be used as general purpose registers.

R14 is used to save the next instruction address when a branch with link instruction is executed.

However, in different modes, the same register number may not correspond to same register. The following paragraph explains which registers can be accessed in the different modes.

User :	R0~R15			
FIQ :	R0~R7	R8_FIQ~R14_FIQ		R15
Supervisor:	R0~R12	R13_SVC	R14_SVC	R15
Abort :	R0~R12	R13_ABT	R14_ABT	R15
IRQ :	R0~R12	R13_IRQ	R14_IRQ	R15
Undefined:	R0~R12	R13_UND	R14_UND	R15

2.6 PSR Register file

The current processor state is saved in the CPSR (Current Processor State Register), the previous processor state is saved in SPSR_XXX (where XXX corresponds to the processor mode). So there are 6 PSR registers total.

The format of PSR registers is show below:

Bit:	Function
31:	Negative
30:	Zero
29:	Carry
28:	Overflow
7:	IRQ disable
6:	FIQ disable
5:	Thumb state
4:0	processor mode

the processor mode in 4:0 is show below:

10000 :	User
10001:	FIQ
10010:	IRQ
10011:	Supervisor
10111:	Abort
11011:	Undefined

2.7 Exceptions

Now there are three types of exceptions supported. They are:

1. Software interrupt (SWI)
2. Fast interrupt request (FIQ)
3. Normal interrupt request (IRQ)

For details of exception, please refer to following chapters.

3.Storage Hierarchy

The storage hierarchy of the nnARM includes several levels. The first level is the instruction prefetch buffer and the load/store component in the MEM stage. The next level is the cache, including instruction cache and data cache. The lowest level is the memory controller. The following figure 3.1 illustrates this:

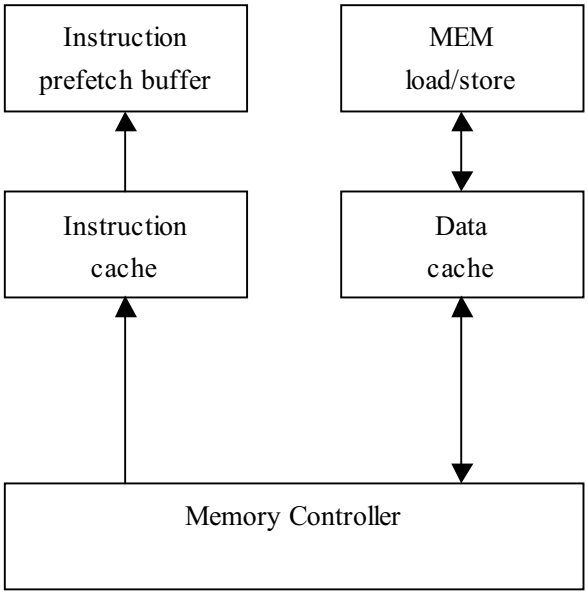

figure 3.1

3.1 Memory controller

The current nnARM does not really implement a memory controller, it only has a behavioral description of one, so therefore a description the memory controller is not very valuable. I must say that I do not know very much about memory and memory controllers. So this memory controller will eventually be replaced by a better memory controller. I will not describe it in detail.

The memory controller has a 32 bit bidirectional databus, a 32 bit address bus input, a read/write flag input, a memory request signal input, a byte/word access flag input, a sequential/non-sequential access mode flag input. And a wait signal output.

In a word, it is very simple and not suited for real applications. Do not pay too much attention to it.

3.2 Instruction cache

The instruction cache has 256 bytes (do not laugh, I do not know how to describe a large array of register words that can be randomly accessed and that can exist in combinational logic). If described as separate registers, the .v file would be too large…

In a large cache, the number of tag fields is also very large, and to

compare each one with the current input address ("set associativity"), I must use combinational logic. The tag fields must be listed in the sensitivity list. But the synthesis tools do not allow room for writing the whole name, it tells me to list all the fields separately.

Who can tell me how to solve it?

The instruction cache has 4 sections, every section contain 4 lines, every line containing 4 words.

The Address[5:4] selects the section, and then the cache controller compares the entire Address[31:6] with each tag field of the 4 lines in this section, if a match is detected then Address[3:2] is used to select the corresponding word in this line.

If no tag field matches the Address[31:6], then the wait signal is brought high to stop the requester and go to the external memory to get the requested cache block.

3.3 Instruction prefetch

The instruction prefetch buffer contains 8 entries. Every entry can contain one 32 bit instruction.

The instruction prefetch buffer has been separated into two parts: the first 4 instructions and the last 4 instructions. When the processor is accessing the first half, the prefetch logic will go to instruction cache to fetch the other half. The other half is filled in like manner.

If the requested address does not fall within the address range of the buffer, then the prefetch logic will enable the wait signal to stop the requester and go to cache to fetch the desired cache block.

3.4 Data cache

The data cache is the same size as the instruction cache.

The data cache has 4 sections, every section containing 4 lines, every line containing 4 words.

The Address[5:4] selects the section, and then the cache controller compares the entire Address[31:6] with each tag field of the 4 lines in this section. If a match is found, then Address[3:2] is used to select the corresponding word in this line.

If a cache miss occurs, then the controller will determine if there is a blank line in this section,

if so, then it will go to memory to fetch the desired cache block into this line.

If not, then it will see if there is a line that is not dirty,

if yes, then it will go to memory to fetch desired cache block into this line.

If not, it will select a random line to write back to memory and then read in the desired cache block into this line.

3.5 A word about future cache and memory systems

To use nnARM in a real application environment, a suitable cache and memory system must be developed. So we developed a new model for future development, shown in Figure 3.2.

First, the nnARMCore will be separate from the cache and memory system. It uses a very simple interface to access the Bus2Core module. It treats Bus2Core as a register file with wait signal.

Second, Bus2Core acts as a protocol translator between nnARMCore and the special type of bus (such as AMBA CoreConnect or WishBone).

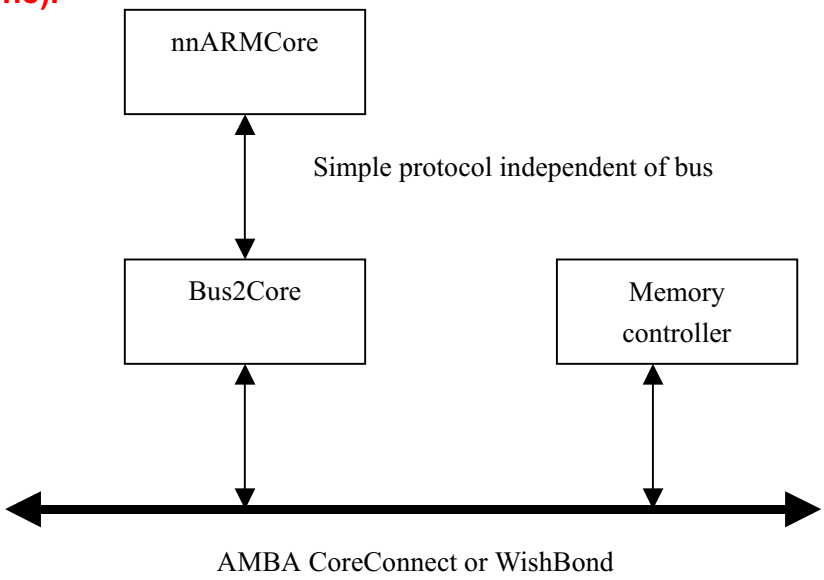

figure 3.2

Third, There can be many possible types of Bus2Core.

For the most simple case, Bus2Core can just translate signals between nnARMCore and the bus. No Cache, no buffer. The greatest advantage of this type of implementation is: SIMPLICITY. This will make it very suitable for simple applications that do not require high performance. But for high performance applications, it is not a good choice.

For a complex case, it can contain cache, MMU (memory management unit), TLB (translation lookaside buffer) and so on. This will make it support most advanced features found in modern processors.

This module is currently under development by Mian Yousaf from Pakistan.

4 Pipeline Overall description

This chapter will describe the overall pipeline structure.

4.1 Pipeline interlock

Because nnARMCore is a multiple stage pipeline design, there may be some conditions which make the pipeline stall (such as a load instruction waiting for its result from memory). In such a case the instructions that precede the stalled instruction must continue to run and write their results to registers. But all instructions that follow the stalled one (including this stalled instruction) must stop to wait.

So a pipeline interlock has been implied for use in nnARMCore. Every stage of the pipeline must include some I/O port to support pipeline interlocking. These ports are listed below:

In_CanNextStageGo This signal comes from the next pipeline stage's Out_CanIGo output port. It indicates whether that next stage can continue to run or not.

Out_CanIGo This signal goes to the previous pipeline stage's In_CanNextStageGo. It tells the previous stage that whether this stage can continue to run or not.

When a stage stalls, it will make Out_CanIGo==1'b0. This will make the previous stages stall. So a stall in one stage will make all its previous stage to stall.

Various stages have various names for these two ports:

	In_CanNextStageGo	Out_CanIGo
IF	in_IDCanGo	No
ID	in_ALUCanGo	in_IDOwnCanGo
ALU	in_MEMCanGo	out_ALUOwnCanGo
MEM	No	out_MEMOwnCanGo

These signals make the entire pipeline run in a consistent way.

4.2 Pipeline register refresh

Because nnARMCore is a multiple stage pipeline design, when an instruction has computed its result it cannot write to the register file immediately. It must wait until it reaches the MEM stage. But before it writes its result, another instruction that needs this result may have begun to run. So if we do not take care of this case, then the latter instruction will read an erroneous register value. The following example illlustrates this idea:

```
1       ADD r0,r1,r2        //r0=r1+r2
2       ADD r3,r0,r1        //r3=r0+r1
```

When 1 reaches the end of the ALU stage, it has got the new value for r0, but has not written it to r0. (It must wait until it reaches the end of the MEM stage to update r0.) But now instruction 2 has reached the end of the ID stage and is about to be latched in the pipeline register of the ALU stage. If we do not take care of this, instruction 2 will use the old value of r0 from the register file and compute an erroneous value for r3.

A forwarding feature has been built to deal with cases of data forwarding. Assume we are now dealing with the left operand of current instructions, and the left operand comes from Rx (of course It may also come from an immediate value-- in that case Rx has no meaning.)

```
If (Left operand comes from an immediate value)
        Directly read the left operand bus
Else if (ALU stage main thread has Rx as its target)
        Read ALU main thread's result
Else if (ALU stage simple thread has Rx as its target)
        Read ALU simple thread result
Else if (MEM stage main thread has Rx as its target)
        Read MEM main thread result
Else if (MEM stage simple thread has Rx as its target)
        Read MEM simple thread result
Else
        Read left operand bus for register value
```

The refresh of the right operand and the third operand is done in the same way.

The CPSR and SPSR registers must also be refreshed in the pipeline similarly. And when any component of nnARM needs a condition flag (such as carry, negative, zero, overflow, interrupt mask, fast interrupt mask and thumb state), they also need the fresh values instead of the old values in the status register file.

5 Instruction Fetch Pipeline Stage(IF)

The IF stage performs the following operations:

1. Increment PC to next instruction address normally.
2. Deal with branch requests from the ALU or MEM stage.
3. Send out PC to fetch instructions from the prefetch buffer.

These operations are described in the following section.

5.1 Increment PC

The IF logic has 1 register read port and 1 register write port from the general register file. These ports are always active and the register number is set to R15 for these ports.

A simple Adder increments the PC value coming from the read port, and sends the result to the write port.

5.2 Branch

When a branch instruction or an ALU instruction with the PC as its destination reaches the ALU stage, it requires a change of PC.

At the same time, if a load instruction with the PC as its destination reaches the MEM stage, it also requires a change of PC.

The request coming from MEM will be processed first, the address from MEM will be loaded into PC. At the same time, a signal is sent to all pipeline stages between IF and MEM to clear those pipeline registers. This is because a load to PC instruction means the instructions following the load instruction will not be executed.

If there is no request from the MEM stage, and the ALU stage wants to change the PC, then all stages between IF and ALU will be cleared. And the address from ALU will be sent to PC.

5.3 Fetch Instruction

The IF sends the PC value from the register read port to the prefetch buffer logic.

At the positive edge of the clock, if the wait signal from prefetch buffer is active, then it means the prefetch buffer can not satisfy the request now, and the IF must force a blank instruction into the pipeline and continue to wait.

If the wait signal from prefetch buffer is not active, IF reads the instruction and feeds it to the decoder.

The following diagram illustrates this.

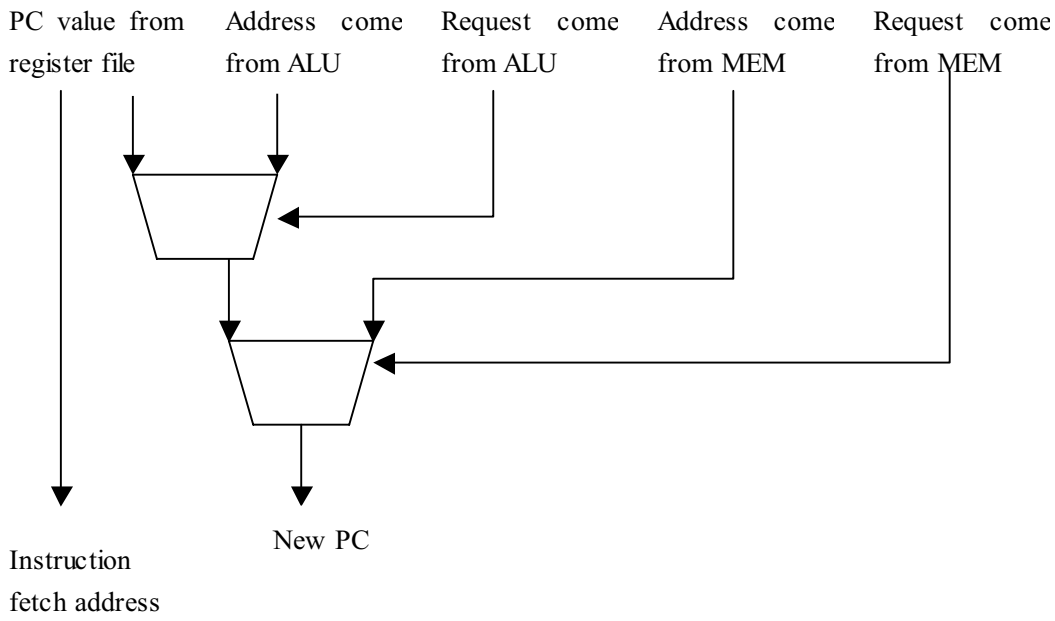
PC value from register file
Address come from ALU
Request come from ALU
Address come from MEM
Request come from MEM
New PC
Instruction fetch address

figure 5.1

6 Decoder for ARM Instruction Set(ID)

This release now supports both the Thumb instruction set and the normal ARM instruction set.

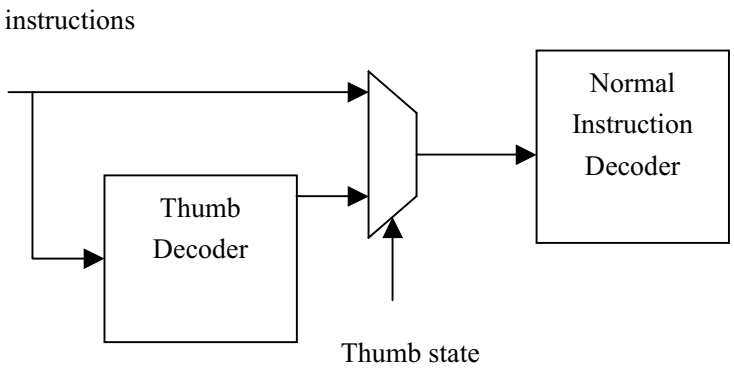

Figure 6.1

The Thumb state bit in CPSR selects either instructions coming directly from the fetch stage or instructions coming from the Thumb decoder.

The ID stage decodes an ARM instruction into micro operations to the ALU and MEM stages.

The ALU and MEM stages both have three threads (this "thread" is not the same as multiple thread processor). One main thread, one simple thread and one PSR thread. The following figure illustrates more clearly:

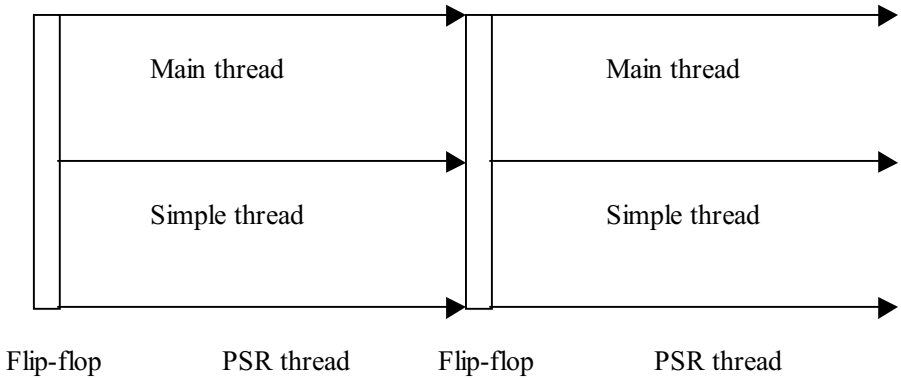

Figure 6.2

The main thread performs all computations in the ALU, and performs all load/store operations in the MEM stage. Finally, it performs

the first register write operation.

The simple thread performs simple data selection in the ALU stage, and performs the second register write operation (if it exists) in the MEM stage.

The PSR thread performs PSR register file write operations.

You will find that only the main thread can stall the pipeline, because it contains load/store and the most complex operations such as multiply (The multiply operation is currently done in 1 clock cycle. This seriously lengthens the clock period, and results in a slow down of the clock frequency. I will modify it in the future to be a "pipelined multiplier unit" which will not seriously slow down the clock, because it will not try to complete the entire multiply in one clock cycle).

When the main thread stalls, then all threads of all stages behind it stall at the same time.

For details of ALU and MEM stage operation, please refer to the next two chapters.

The following instructions are supported:

1. multiply(MLA) and multiply then add(MLA)
2. branch(B) and branch with link(BL)
3. PSR transfer(MRS and MSR)
4. all ALU instructions
5. single data transfer(LDR/STR)

The following are the instructions not yet supported:

1. single data swap(SWP)
2. block data transfer(LDM/STM)
3. all coprocessor instructions
4. software interrupt(SWI)

I will describe how to decode these supported instructions into micro operations, only the main thread and simple thread are included in the following figure. The psr thread is described in a separate section:

Instruction type	ALU main thread	ALU simple thread	MEM main thread	MEM simple thread
MUL	ALUType_Mul	ALUType_Null	MEMType_Mov Main	MEMType_Null
MLA	ALUType_Mla	ALUType_Null	MEMType_Mov Main	MEMType_Null
B	ALUType_Add	ALUType_Null	MEMType_Null	MEMType_Null
BL	ALUType_Add	ALUType_MvN extInstruction Address	MEMType_Null	MEMType_Mo vSimple
MRS	ALUType_Null	ALUType_MvS PSR or ALUType_MvC PSR	MEMType_Null	MEMType_Mo vSimple
MSR	No operation except for PSR thread			

ALU instruction	Corresponding ALU operation	ALUType_Null	MEMType_Null (for tst, teq,cmp,cmn) MEMType_Mov Main(for other case)	MEMType_Null
LDR	ALUType_Add or ALUType_Sub depend on type of address calculate	ALUType_Mvl when post index is required	Varies type of load	ALUType_Mov Main when write back is required
STR	ALUType_Add or ALUType_Sub depending on type of address calculate	ALUType_Mvl when post index is required	Varies type of store	ALUType_Mov Main when write back is required

6.1 Operand preparation

The decoder has the duty to send out register read requests to the register file and send out immediate values. This is the so called operand preparation feature.

In the most serious case, an instruction may require 3 operands. For example, an ALU instruction that involves a shift count from a register.

So the decoder has 3 read channels. I call them first, second and third read channels. Every channel has the following signals:

1. **"read register enable" (goes to the register file) When the operand comes from a register, this signal will be high, else it will be low**
2. **"read register number" (goes to the register file)**
3. **"forwarding disable" (goes to the ALU stage) If this signal is true, then forwarding will not be performed on this channel, else forwarding will use the "freshest" value of this register from the pipeline.**
4. **"read out bus" (from the register file to the ALU stage) When this operand comes from the register file, this bus carries the corresponding register contents, else this bus carries the immediate value from the decoder.**

The following figure illustrates this more clearly:

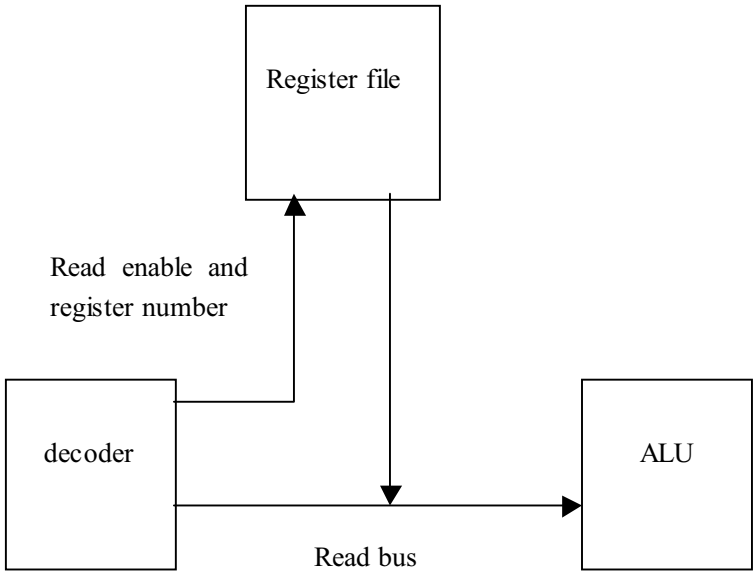

figure 6.3

The PC is a special case. The PC is never read directly from a register by the decoder. It always goes with the corresponding instruction. That is to say, any instruction going from IF stage to decoder stage will carry its own PC. The PCs in the pipeline are never affected by forwarding.

6.2 PSR thread

Only the following ALU instructions may change the PSR register file: MUL,MLA and MRS.

In these ALU instructions, there is a S bit that indicates whether the instruction writes to CPSR.

If the S bit is set, and the destination register is not the PC, then it must write to CPSR. Therefore, the decoder generates the ALUPSRType_WriteConditionCode and MEMPSRType_WriteConditionCode micro operations for the PSR thread.

If the S bit is set, but the destination is the PC, then ALUPSRType_SPSR2CPSR and MEMPSRType_WriteCPSR micro operations are generated by the decoder for the PSR thread.

If S bit is not set, no PSR registers are written.

The S bit has the same feature for the MUL and MLA instructions except that the PC can not act as destination of MUL and MLA.

MRS moves a general purpose register to CPSR or SPSR. Corresponding micro operations are generated.

6.3 Signal "out_ALUMisc"

This signal performs some special functions.

out_ALUMisc[31:28]: this field contains the condition code of this instruction. The ALU stage uses this code to decide whether this instruction qualifies under the current processor state and can continue to run.

Out_ALUMisc[0]: when decoding a normal instruction, the third read channel is used to carry the shift count from the register file or as an immediate value. But when decoding a store instruction, the shift count is always a 5 bit wide immediate value, and at the same time the stored value occupies the third channel. So, under this condition Out_ALUMisc[0] is high, which puts the shift count in Out_ALUMisc[5:1].

Out_ALUMisc[6]: when decoding a branch or an ALU instruction that wants to modify the PC, Out_ALUMisc[6] is high.

Out_ALUMisc[7]: when decoding a load to PC, this is high.

6.4 Bubble insertion

In some special cases, the decoder must insert a "bubble" into the pipeline, or the nnARM will not run correctly.

When there is a load to a general purpose register Rn, and the following instruction wants to use Rn as its source operand, then if the following instruction dispatches to the ALU immediately, it will miss the freshest value of Rn. This is because when it dispatches, the load instruction is still in the ALU, and is not yet finished.

After inserting a bubble, the decoder must wait until that bubble goes to the MEM stage. This means that the load instruction has finished loading and is ready to forward Rn to the following instruction.

6.5 Interrupt Handling

This release now supports interrupts, including normal interrupts and "fast interrupt requests."

When an interrupt arises, a special instruction is inserted into the decoder to generate all the signals that are required to process it. These signals are very similar to those of the SWI instruction.

Instruction type	ALU main thread	ALU simple thread	MEM main thread	MEM simple thread
SWI,IRQ,FIQ	ALUType_Mov	ALUType_MvNextInstructionAddress	MEMType_Null	MEMType_MovSimple

In all these three types of interrupts, the ALU main thread is used to branch to an interrupt handler written in software. And the MEM main

thread is not used.

The ALU and MEM simple threads are used to write the return address to link register (R14).

The PSR thread is used to write the old CPSR into SRSR, and to write the new CPSR into the CPSR register.

7 ALU stage

The ALU stage contains three threads: the main thread, the simple thread and the PSR thread. The following sections describe them.

7.1 Main thread

This thread performs all computations. It supports the following micro operations:

ALUType_Add	LeftOperand + RightOperand
ALUType_Sub	LeftOperand – RightOperand
ALUType_And	LeftOperand And RightOperand
ALUType_Eor	LeftOperand Eor RightOperand
ALUType_Rsb	RightOperand - LeftOperand
ALUType_Adc	LeftOperand + RightOperand + Carry
ALUType_Sbc	LeftOperand – RightOperand + Carry - 1
ALUType_Rsc	RightOperand – LeftOperand + Carry - 1
ALUType_Tst	As And, but do not write result, only set flags
ALUType_Teq	As Eor, but do not write result, only set flags
ALUType_Cmp	As Sub, but do not write result, only set flags
ALUType_Cmn	As Add, but do not write result, only set flags
ALUType_Orr	LeftOperand Or RightOperand
ALUType_Mov	RightOperand
ALUType_Bic	LeftOperand And ~RightOperand
ALUType_Mvn	~RightOperand
ALUType_Mul	LeftOperand Mul RightOperand
ALUType_Mla	(LeftOperand Mul RightOperand) + ThirdOperand

All these micro operations are sent to the “ALUComb” module.

7.2 ALUComb

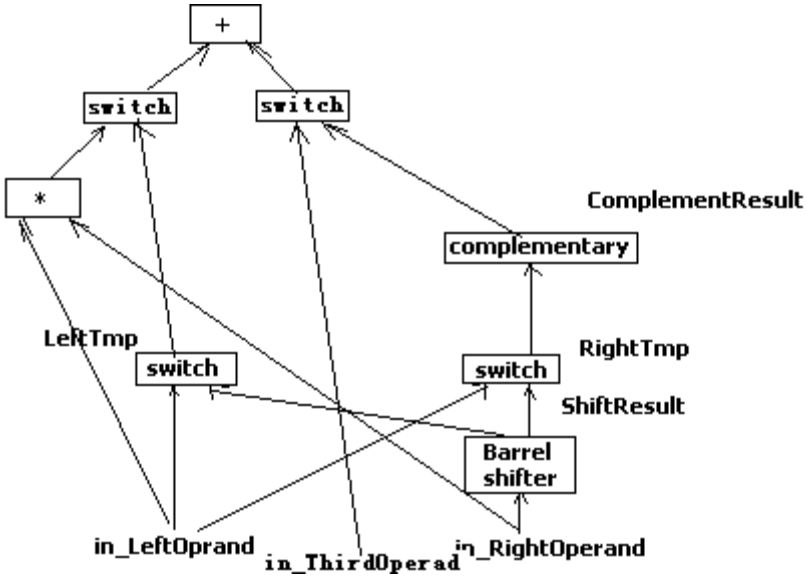

figure 7.1

Figure 7.1 shows the structure of the ALUComb module.

ALUComb supports all ALU operations in the ARM instruction set.

The two main inputs are in_LeftOperation and in_RightOperation. The in_RightOperation is shifted by the Barrel shifter.

Then, the two switches (or multiplexers) select which will be the left operation and which will be the right one. This is needed because the ARM instruction set includes operations of the form: operand1 op operand2, and also conversely of the form: operand2 op operand1.

Next, to deal with the SUB operation, the twos-complement of RightTmp is produced.

After that, the two operands are sent to the adder. Other types of operands such as sub, rsb, sbc and adc are similar to this.

Recently MLA support has been added. Now the two switches near the adder are used to select which operand will be added when doing "multiply and add."

The logic operations are very easily done and are not described here.

7.3 Simple thread

This simple thread is used to perform some simple operations. It supports the following micro operations:

ALUType_Mvl use left operand as simple thread output
ALUType_Mvr use right operand as simple thread output
ALUType_MvCPSR use CPSR as simple thread output
ALUType_MvSPSR use SPSR as simple thread output
ALUType_MvNextInstructionAddress
Use address of next instruction as simple thread output

7.4 PSR thread

The PSR thread performs its computation and writes to the PSR file.

At the same time, because all ARM instructions have a conditional execution field, all instructions must use forwarding to get the freshest CPSR status to decide whether they can continue to run before entering the ALU stage.

7.5 Forwarding

The following program explains how general purpose register forwarding is performed. Assume that Rn is the source operand:

```
if(operand comes from immediate value)
        read it from corresponding read bus
else if(current ALU main thread wants to write Rn)
        forward result from current ALU main thread
else if(current ALU simple thread want to write Rn)
        forward result from current ALU simple thread
else if(current MEM main thread want to write Rn)
        forward result from current MEM main thread
else if(current MEM simple thread want to write Rn)
```

forward result from current MEM simple thread
else
read it from corresponding read bus

Following program explains how CPSR register forwarding is performed:

```
If(CPSR is from the current immediate value)
    Read in from read bus
Else if(current ALU want to write CPSR)
    Read it from ALU
Else if(current MEM want to write SPSR)
    Read it from MEM
Else
    Read in from read bus
```

SPSR forwarding is similar.

7.6 Processing the condition field

The most significant 4 bits of an instruction indicate under what conditions the instruction can continue to run and write its result.

I use the CPSR processed by forwarding to determine if the instruction can continue to run.

0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

If an instruction can not continue to run, a bubble is inserted into the ALU stage and this instruction will disappear. A conditional branch is also performed in this way.

7.7 Branches

For a branch instruction, it generates a branch request to all stages between IF and ALU, all stage are cleared by this request signal.

It also sends out the branch destination address, the IF must restart to fetch at that address.

8 MEM stage

The MEM stage contains three threads: the main thread, the simple thread and the PSR thread.

The main thread performs all load/store operations and writes loaded values into registers.

The simple thread performs the second register writing(for example a write back of base address register in load/store instructions).

The PSR thread performs writes to the PSR register file.

8.1 Main thread

The main thread supports the following micro operations:

MEMType_MovMain	**write main ALU thread result to register**
MEMType_MovSimple	**write simple ALU thread result to register**
MEMType_LoadMainWord	**use main ALU thread result as address to load a word**
MEMType_LoadMainByte	**use main ALU thread result as address to load a byte**
MEMType_LoadSimpleWord	**use simple ALU thread result as address to load a word**
MEMType_LoadSimpleByte	**use simple ALU thread result as address to load a byte**
MEMType_StoreMainWord	**use main thread ALU result as address to store a word**
MEMType_StoreMainByte	**use main thread ALU result as address to store a byte**
MEMType_StoreSimpleWord	**use simple thread ALU result as address to store a word**
MEMType_StoreSimpleByte	**use simple thread AUL result as address to store a byte**

8.2 Simple thread

The simple thread supports the following micro operations:

MEMType_MovMain	**write main ALU thread result to register**
MEMType_MovSimple	**write simple ALU thread result to register**

8.3 PSR thread

The PSR thread supports the following micro operations:

MEMPSRType_WriteSPSR	**write freshest SPSR in pipeline to SPSR register**
MEMPSRType_SPSR2CPSR	**write freshest SPSR in pipeline to CPSR register**

MEMPSRType_WriteCPSR write freshest CPSR in pipeline to CPSR register
MEMPSRType_WriteConditionCode write condition code only to CPSR register, because of forwarding, this is the same as MEMPSRType_WriteCPSR

8.4 Change of PC

When the current micro operation is a load to the PC, then after the load is finished, the new PC must be sent to the IF stage. At the same time, a branch request must be sent to all stage between IF and MEM to clear them.