nnARM Architecture Specification

version 1.11

By ShengYu Shen
From NUDT
2001.6.10

NOTE

This documentation describe the architecture of the nnARM processor core. Every main release of the nnARM have only one such documentation. Any change over nnARM v2.XX will not include in this documentation. Please refer to newer version and the comment in source code.

Release Log

V1.00	**2001.4.11**
V1.10	**2001.6.1**
V1.11	**2001.6.10**

1.Introduction

The nnARM project is a development project start at 2001.3.24.The purpose of this project is to develop a synthesable high performance embedded processor core that can run ARM instruction set.

Now, the second main release of this soft core have been complete. It contain the following component :

1. **a behavior description of memory controller**
2. **behavior description of instruction cache controller and data cache controller**
3. **a RTL synthesable instruction prefetch buffer**
4. **a RTL synthesable instruction fetch component**
5. **a RTL synthesable decoder for ARM**
6. **a RTL synthesable full function ALU that can support all kinds of ALU operation of ARM**
7. **a RTL synthesable mem stage that can perform load and store operation**
8. **a RTL synthesable register file**

Note: The Tomasulo structure have been removed because I can not manage to deal with the complexity of design it. At the same time, I think I can not found enough logic and interconnect resource on a FPGA to imply such a complex structure.

This documentation will be organized in the following way: First, in the second section, I will describe the overall architecture of nnARM V1.10. And then, I will describe the storage hierarchy of nnARM V1.10. After that, pipeline will be describe .

2. Overall architecture

2.1 Structure Introduction

The following figure2.1 will tell you about the external view of this processor.

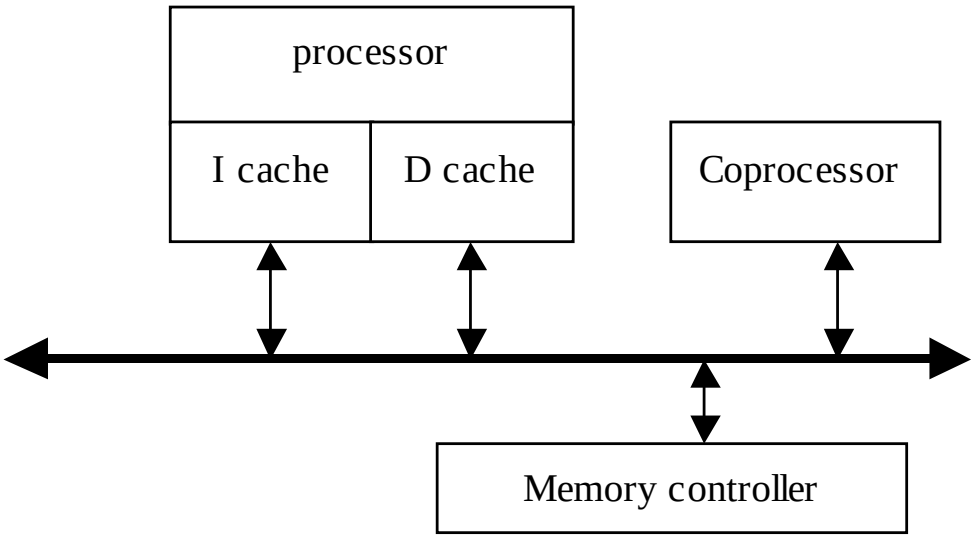


figure 2.1

The processor have separate data cache and instruction cache. Both have been describe in behavior level. The detail of both cache will be given in the next chapter

The memory controller is also in behavior level description. It is external to the chip. So a behavior level description is enough for it.

The coprocessor can accept the request from main processor through the memory bus. It can distinct the memory request and coprocessor request. The memory controller also have this capability. The coprocessor interface have been consider now. Probably I will add it later.

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

The first stage is IF, it fetch one instruction from prefetch buffer every cycle. It is the so called 1 issue pipeline. 2 issue or 4 issue is possible, but is will seriously increase the complexity of overall architecture. And a report from ARM say that a two issue will get a 20%

performance improve only. I think that because ARM instruction set is more like a CISC than common RISC, its code is very density, one instruction word can do more work than same size RISC instruction word. So 1 issue is enough for it.

The second stage is the ID stage. In this stage the decoder will translate the instruction to multiple microinstruction and send them to pipeline structure.

After that the microinstructions go to the ALU stage to perform varies type of computing , include and eor sub rsb add adc sbc rsc tst teq cmp cmn orr mov bic mvn , at the same time a booth multipler is chain with the ALU to perform MUL and MLA operation.

After the ALU stage have perform the correspond operation, it will pass the computed result and the micro operation for MEM stage to MEM stage. In this stage, the load/store operation will go to access memory. At the end of this stage, all result for register file will be write back.

I think I must say some words about the forwarding of pipeline. If a instruction n use Rn as its destination register, and the following instruction n+1 will use Rn as its source operand, when n complete its ALU stage, the result have not been save to Rn, but the n+1 want to use Rn to compute its result, so a forwarding must be perform to pass n result to n+1.

The following figure will show you it more clear.

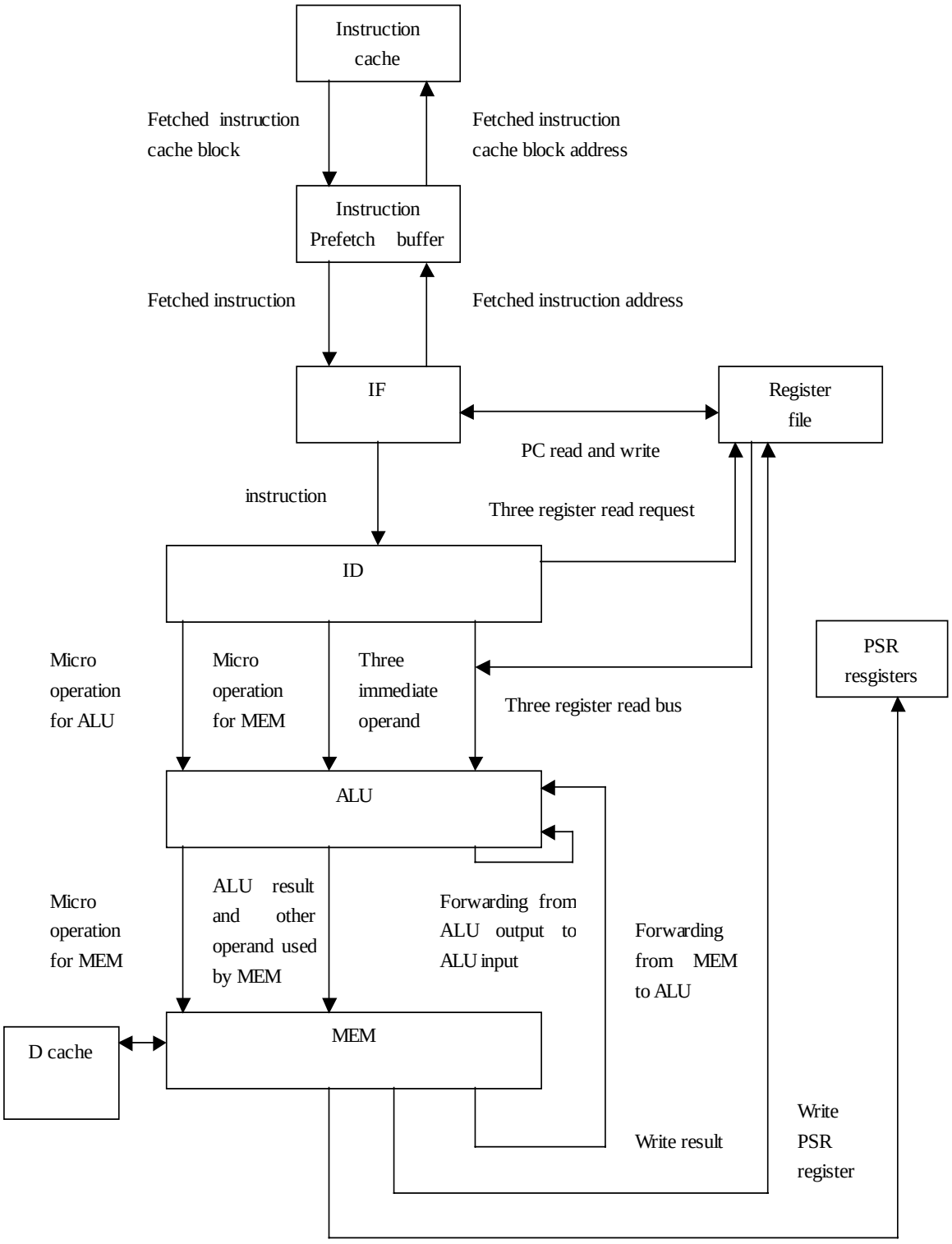

Instruction cache
Fetched instruction cache block
Fetched instruction cache block address
Instruction Prefetch buffer
Fetched instruction
Fetched instruction address
IF
Register file
PC read and write
instruction
Three register read request
ID
PSR resgisters
Micro operation for ALU
Micro operation for 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

This processor current only support little endian. That is to say, in a Word, the least significant byte is at the lowest address, and the address of a word is the address of its least significant byte. The memory organization is show below.

Big endian is not support now, and I do not have the plan to support it.

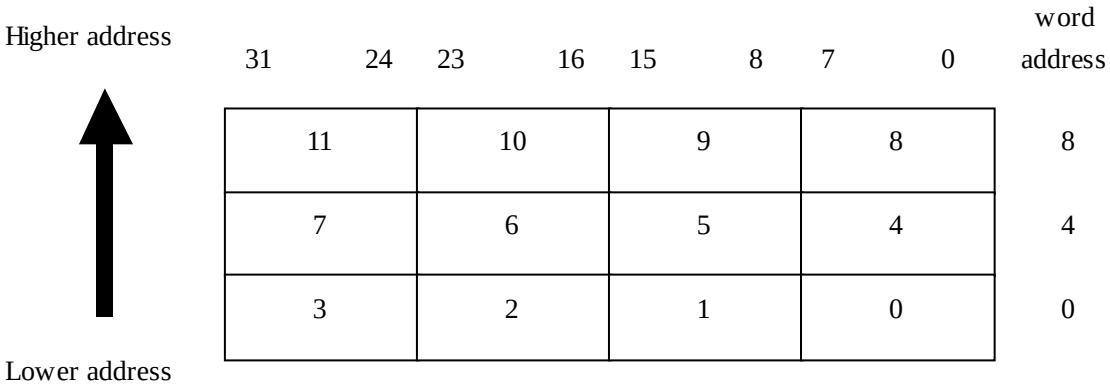


figure 2.3

2.3 Address Bus Width

Because the nnARM is a brand new design, so it do not have the backward compatible problem. So only 32 bit address bus width is support, 26 bit bus mode that used in ARM7 processor will not support here , and I do not have the plan to support it.

2.4 Processor Mode

nnARM processor support the six operation modes:

1) **User mode: the normal program execution mode**
2) **FIQ mode:design to support a data transfer or channel process**
3) **IRQ mode:general purpose interrupt handling**
4) **Supervisor mode:protected mode for OS**
5) **Abort mode:memory fetch failure**
6) **Undefined mode:an undefined instruction executed**

Mode change may controlled by software or external interrupt or exception. Most user program execute in user mode. Other mode are called privileged mode that use to handle interrupt or exception.

2.5 General Register file

The register file now contain 31 general purpose register. Which set of register can be access is depend on the mode of the processor.

In any time, there is 16 register that can be access by software. They are R0 to R15. R15 is program counter, other register can all be used as general register.

R14 is use to save the next instruction address when a branch

with link instruction is executed.

But in deferent mode, the same register number may not correspond to same register. The following paragraph will tell you which register can be access in every mode.

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**
Undifined:	**R0~R12**	**R13_UND**	**R14_UND**	**R15**

2.6 PSR Register file

The state of current processor is save in CPSR register, the old state of varies processor mode is save in SPSR_XXX(XXX correspond to processor mode). So There is total 6 PSR registers.

The format of PSR register is show below:

31: Negative
30: Zero
29: Carry
28: Overflow
7: IRQ disable
6: FIQ disable
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 Exception

I have not support any kinds of exception in current release. I want to support a full ARM7 instruction set first.

BUT I DO NOT THINK IT IS DIFFICULT TO SUPPORT EXCEPTION.

I think exception is another kind instruction, when the decoder detect that there is a exception, it will stop fetch instruction and generate an exception instruction into pipeline to perform varies kinds of operation to change processor mode and go to corresponding vectors.

3.Storage Hierarchy

The storage hierarchy of the nnARM include several level. The first level is the instruction prefetch buffer and the load/store component in the MEM stage. The next level is the cache, include instruction cache and data cache. The most low level is the memory controller. The following figure 3.1 will show you more clear.

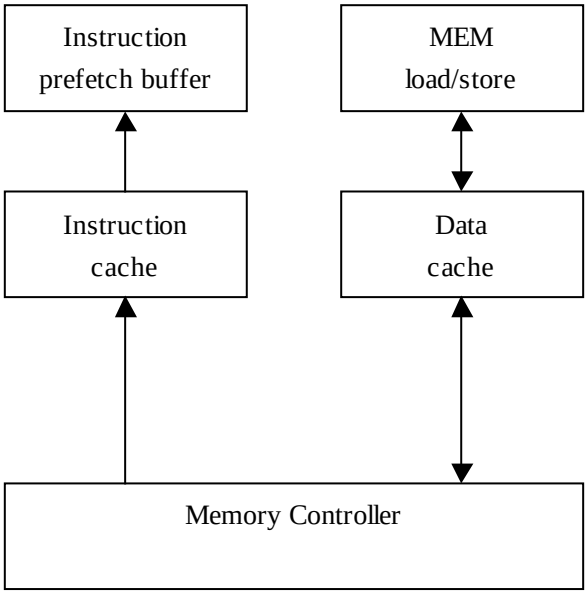


figure 3.1

3.1 Memory controller

This description of the memory controller is not valuable. I must say that I do not know very much about memory and its memory controller. So this memory controller will soon be replace by a more well designed controller. I will not describe its in detail.

The memory controller have a 32 bit bidirection 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. A wait signal output.

In a word, it is too simple and not suite for real application. Do not pay too much attention to it.

3.2 Instruction cache

This instruction cache have 256 byte(do not laugh at it, I do not know how to describe a large array of register words that can be random access and can attend the combinational logic computing). If describe as several separate register, the .v file will be too large.

Because in a large cache, the tag field will be very large too, to compare it with currently input address, I must use combinational logic. The large tag field must be list in the sensitive list. But the synthesis

tools do not allow I write them as a whole name, it tell me to list all the field separately.

Who can tell me how to solve it?

The instruction cache have 4 section, every section contain 4 lines, every line contain 4 words.

The Address[5:4] select the section, and then the cache controller compare the entire Address[31:6] with each tag field of the 4 lines in this section, if found the desired address then use the Address[3:2] to select the correspond word in this line.

If no tag field match the Address[31:6], then make the wait signal high to stop the requester and go to the memory to got the cache block.

3.3 Instruction prefetch

The instruction prefetch buffer contain 8 entry, every entry can contain one 32 bit instruction.

The entire instruction prefetch buffer have been separate in two part: the first 4 instructions and the last 4 instructions. When accessing the first half, the prefetch logic will go to instruction cache to fetch the other half. It is the same case for the second half.

If the request address do not fall in 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 desire cache block.

3.4 Data cache

The data cache is the same size as instruction cache

The data cache have 4 section, every section contain 4 lines, every line contain 4 words.

The Address[5:4] select the section, and then the cache controller compare the entire Address[31:6] with each tag field of the 4 lines in this section, if found the desired address then use the Address[3:2] to select the correspond word in this line.

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

if yes, 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.

4 Instruction Fetch Pipeline Stage(IF)

The IF stage perform the following operation.

1. **increase pc to next instruction address at normal condition**
2. **deal with branch request come from ALU or MEM stage.**
3. **send out PC to fetch instruction from prefetch buffer**

I will describe them at following section

4.1 Increase PC

IF have 1 register read port from general register file. This port is always active and the register number to read is always R15.

At the same time, IF have a register write port, This port is always active and the register number to write is always R15.

I use a simple Adder to increase PC value come from the read port , and send result to write port.

4.2 Branch

When a branch instruction or an ALU instruction with PC as its destination reach ALU stage, it will require a change of PC.

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

The request come from MEM will be process first, the address come from MEM will be send to PC. At the same time, a signal will send to all stage between IF and MEM to clear there pipeline register. Because if a load to PC instruction occur, all instruction following the load instruction will not be executed.

If there is no request from MEM stage, and ALU stage want to change PC, then all stage between IF and ALU will be clear. And the address come from ALU will be send to PC.

4.3 Fetch Instruction

The IF will send out PC value read from the register read port to prefetch buffer.

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 pipeline and continue to wait.

If the wait signal from prefetch buffer is not active, IF can read in the instruction. And feed it to decoder.

The following diagram will show you all together.

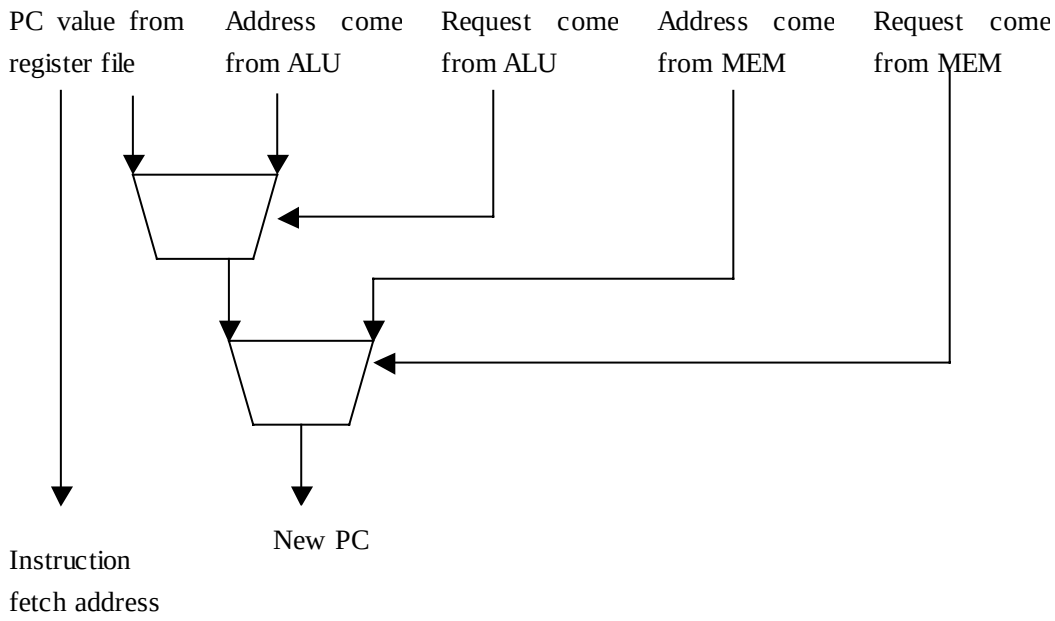

PC value from register file
Address come from ALU
Request come from ALU
Address come from MEM
Request come from MEM
New PC
Instruction fetch address

figure 4.1

5 Decoder for ARM Instruction Set(ID)

ID stage just decode the instruction into micro operation to ALU and MEM stage.

The ALU and MEM stage both have three thread(this "thread" is not same as multiple thread processor). One main thread, one simple thread and one PSR thread. Following figure will show you more clear.

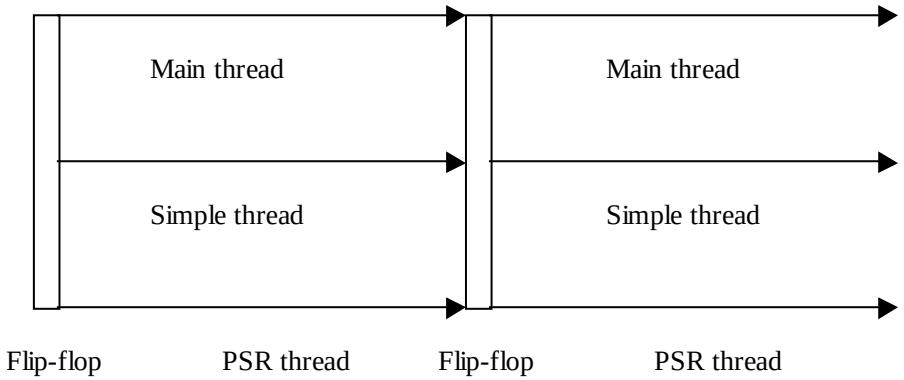


Figure 5.1

Main thread perform all computation in ALU, and perform all load/store in MEM stage, finally perform the first register write operation.

Simple thread perform simple data selection in ALU stage , and perform the second register write operation(if exist) in MEM stage.

The PSR thread perform PSR register file write operation.

You can found that, only the main thread can stall the pipeline, because it contain load/store and most complex operation such as multiple(now the multiple operation consume only 1 cycle, this serious slow down the clock frequency, I will modify it to consume several cycle at future).

When the main thread stall, then all thread of all stage behind it will be stall at the same time.

For detail of ALU and MEM stage operation, please refer to the next two chapter.

Following are instruction supported:

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

Following are instruction unsupported yet:

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

I will describe how to decode these support instruction into micro operation, only main thread and simple thread will be include in following figure, the psr thread will be describe in a separate section:

Instruction type	ALU main thread	ALU simple thread	MEM main thread	MEM simple thread
MUL	ALUType_Mul	ALUType_Null	MEMType_MovMain	MEMType_Null
MLA	ALUType_Mla	ALUType_Null	MEMType_MovMain	MEMType_Null
B	ALUType_Add	ALUType_Null	MEMType_Null	MEMType_Null
BL	ALUType_Add	ALUType_MvNextInstructionAddress	MEMType_Null	MEMType_MovSimple
MRS	ALUType_Null	ALUType_MvSPSR or ALUType_MvCPSR	MEMType_Null	MEMType_MovSimple
MSR	No operation except for PSR thread			
ALU instruction	Correspond ALU operation	ALUType_Null	MEMType_Null(for tst, teq,cmp,cmn) MEMType_MovMain(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_MovMain when write back is required
STR	ALUType_Add or ALUType_Sub depend on type of address calculate	ALUType_Mvl when post index is required	Varies type of store	ALUType_MovMain when write back is required

5.1 Operand preparation

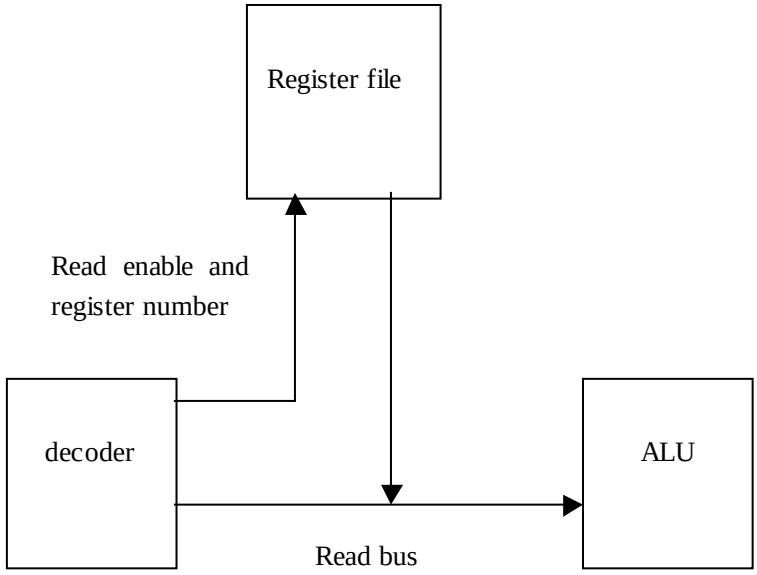
The decoder have the duty to send out register read request to register file and send out immediate value. This is the so call operand preparation feather.

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

So the decoder have 3 read channel. I call them first ,second and third read channel. Every channel have following signal:

1. **read register enable, go to register file, when this operand come from register, this signal will be high, else it will be low**
2. **read register number, go to register file**
3. **if this operand is a immediate value, go to ALU stage, if this signal is true, then the forwarding will not be perform on this channel, else forwarding will use the most fresh value of this register from pipeline, if there is an instruction want to write to this register and have got its result but have not write to register.**
4. **the read out bus from register file to ALU stage, when this operand is come from register file, this bus will carry the corresponding register content, else this bus will carry the immediate value from decoder.**

Following figure will show you more clear



There is a special case, it is the PC. The PC never read directly from register by decoder. It is always go with the corresponding instruction. That is to say, any instruction go from IF stage to decoder stage will carry its own PC. The PC in pipeline never affect by forwarding.

5.2 PSR thread

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

In every ALU instructions, there is a S bit that indicate whether

this instruction may write to CPSR.

If the S bit is set, and the destination register is not PC, then it must write CPSR, the decoder will generate ALUPSRType_WriteConditionCode and MEMPSRType_WriteConditionCode micro operation for PSR thread.

If S bit is set, but destination is PC, then ALUPSRType_SPSR2CPSR and MEMPSRType_WriteCPSR will be generate by decoder for PSR thread.

If S bit is not set, no PSR register will be writen.

The S bit have the same feather for MUL and MLA instruction except that PC can not act as destination of MUL and MLA.

MRS move a general purpose register to CPSR or SPSR. Corresponding micro operation will be generate.

5.3 Signal “out_ALUMisc”

This signal perform some special feather.

out_ALUMisc[31:28]: this field contain condition code of this instruction, the ALU stage will use this field to decide whether this instruction satisfy current processor state and can continue to run.

Out_ALUMisc[0]: when decoding a normal instruction, the third read channel will be use to carry shift count, from register file or as a immediate value. But when decoding a store instruction, the shift count is always a 5 bit width immediate value, at the same time the stored value occupy the third channel, so I make Out_ALUMisc[0] high and send out shift count in Out_ALUMisc[5:1].

Out_ALUMisc[6]: when decoding an branch or an ALU instruction that want to modify PC, I will make Out_ALUMisc[6] high.

Out_ALUMisc[7]: when decoding a load to PC, I will make it high

5.4 Bulk insertion

In a special case, decoder must insert a bulk into pipeline, or the nnARM will not run correctly.

When there is a load to Rn, Rn is a general purpose register, and the following instruction want to use Rn as its source operand. In this case, if the following instruction dispatch to ALU immediately, then it will miss most fresh value of Rn from memory. Because when it dispatch, the load instruction is still in ALU, it have not got its result from memory.

After insert a bulk, the decoder must wait until that bulk go to MEM, this means that the load instruction have finish loading and ready to forward Rn to the following instruction.

6 ALU stage

The ALU stage contain three thread: main thread, simple thread and PRS thread. I will describe they at following sections.

6.1 Main thread

This thread perform all computation. It have following micro operation:

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**
ALUType_Teq	**As Eor, but do not write result**
ALUType_Cmp	**As Sub, but do not write result**
ALUType_Cmn	**As Add, but do not write result**
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 operation will be send to a module named "ALUComb". The detail description of ALUComb is at next section.

6.2 ALUComb

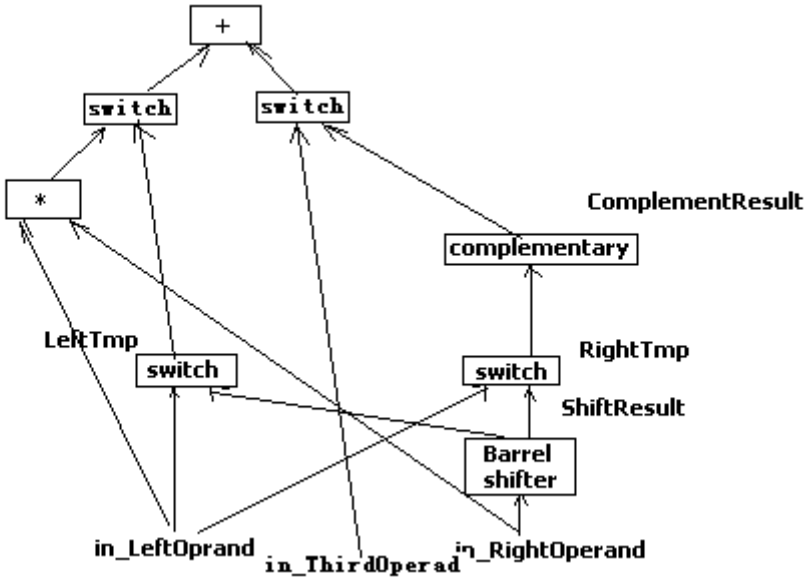


figure 6.1

The figure 6.1 describe the structure of this ALUComb module.

This ALUComb can support all ALU operantion in ARM instruction set.

First, the two input is in_LeftOperation and in_RightOperation. The in_RightOperation is first shift by the Barrel shifter.

And then, the two switch select who will be the left operation and who will be the right one. Because the ARM ALU operation can deal with two types of operation:opeand1 op oprand2 and oprand2 op oprand1. So selection is needed.

After that, to deal with sub operation, the complementary of RightTmp is produce.

After that, the two operand can be send to adder. Other types of operand such as sub,rsb,sbc and adc is similar to this.

Recently I add MLA support in it, now the two switch near adder is use to select which operand will be added.

The logic operation can be done very easy and will not describe here.

6.3 Simple thread

This simple thread is use to perform some simple operation. It has following micro operation:

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

6.4 PSR thread

The PSR thread perform the computation and write of PSR file.

At the same time, because all ARM instructions have a conditional execution field, so all instruction must use forwarding to get most fresh CPSR status to decide whether it can continue to run before it enter ALU stage.

6.5 Forwarding

The following program will tell you how general purpose register forwarding is perform, assume that Rn is the source operand:

```
if(operand come from immediate value)
        read it from corresponding read bus
else if(current ALU main thread want 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 will tell you how CPSR register forwarding perform:
If(CPSR of current is from 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.

6.6 Process the condition field

The most significant 4 bit of an instruction indicate that under what condition can this instruction continue to run and write its result.

I use the CPSR processed by forwarding to determine if this 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 bulk will be insert to ALU stage and this instruction will disappear. A conditional branch is also perform in this way.

6.7 Branch

For an branch instruction, it will generate a branch request to all stage between IF and ALU, all stage will be clear by this request signal.

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

7 MEM stage

The MEM stage contain three thread: main thread, simple thread and PSR thread.

The main thread perform all load/store operation and write loaded value to register.

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

The PSR thread perform the write to PSR register file.

7.1 Main thread

The main thread have following micro operation:

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**

7.2 Simple thread

The simple thread have following micro operation:

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

7.3 PSR thread

The PSR thread have following micro operation:

MEMPSRType_WriteSPSR	**write most fresh SPSR in pipeline to SPSR register**
MEMPSRType_SPSR2CPSR	**write most fresh SPSR in pipeline to CPSR register**

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

7.4 Change of PC

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