UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
DISCIPLINA: LINGUAGEM ASSEMBLY
PROFESSOR GUTO

ESTUDO SOBRE PROCESSADOR ARM7

ALUNOS: ERICH SILVESTRE
PEDRO BACHIEGA

FLORIANÓPOLIS, 12 DE FEVEREIRO DE 2007.

1 Características do Processador ARM7

A arquitetura ARM é uma arquitetura de uso geral em processadores RISC 32 bits que é extensamente usada em muitas aplicações. Ela é projetada para ser programada em linguagem Assembly e tem diversas características avançadas que fazem ela ser extremamente eficiente.

O ARM7 é um processador RISC, com *pipeline* de três estágios e arquitetura de registradores *load/store*. O processador estará sempre ocupado em executar três instruções em diferentes estágios. Enquanto busca a primeira, decodifica a segunda e executa a terceira. Essa arquitetura de *pipeline* é bem simples e evita os conflitos entre estágios de arquiteturas mais avançadas.

Algumas características:

- Os deslocamentos de uma constante são livres.
- Checagem condicional e suas execuções são livres.
- Qualquer número de registradores pode ser transferido para ou da memória usando uma simples instrução.
- Instruções *load/store* tem pré/pós indexamento por constantes ou um segundo registrador com um deslocamento opcional.
- O PC (*program counter*) é mapeado para o registrador 15.

Resultados:

- Código com decimais de ponto fixo rodam na mesma velocidade que códigos de inteiros.
- Poucos ou nenhum *pipeline stall* e nenhum *branching* mal em torno de apenas poucas instruções.
- Alta densidade de código e rápida transferência de memória sem cache.
- Código compacto, rápido e legível.
- Alta simplicidade para fazer o código do PC relativo sem perda na eficiência.

Todo processamento de dados é feito utilizando os registradores da CPU, isto implica que se deve carregar da memória toda variável que for utilizada para processamento.

O processador ARM7 possui sete diferentes modos de execução. Partes desses modos são usados para tratamento de exceções e de interrupções. Um sistema operacional pode utilizar os modos privilegiados para executar códigos de sistema e deixar as aplicações restritas no modo usuário.

Abaixo segue uma pequena descrição de cada modo.

Modo	Descrição
Usuário	Execução normal de programas
Sistema	Executa rotinas privilegiadas do Sistema Operacional
Supervisor	Modo protegido para o Sistema Operacional
IRQ	Tratamento de interrupções comuns
FIQ	Tratamento de interrupções rápidas
Abort	Usado para implementar memória virtual ou proteção de memória
Indefinido	Suporta a emulação em software de co-processadores

A arquitetura do ARM7 define três diferentes perfis de processador: o perfil A para sistemas operacionais sofisticados, ou baseados em memória virtual e aplicações de usuário; o perfil R para sistemas de tempo-real; e o perfil M otimizado para microcontrolador e aplicações de baixo custo.

Todos perfis da arquitetura ARM7 implementam a tecnologia *Thumb® -2* de compressão de código. A arquitetura também inclui as extensões da tecnologia *NEON™* para aumentar o DSP (*Digital Signal Processor*) e o processamento digital por até 400%, e oferece melhor suporte à ponto flutuante no endereçamento que a nova geração de gráficos 3D e física de jogo precisam, assim como aplicações tradicionais de controle embarcadas.

Alguns modos possuem registradores específicos o que facilita a troca de contexto entre eles. Sobre alguns deles sabemos que o CPSR, por exemplo, é o registrador de estado do processador ARM.

O modo THUMB é um modo especial que transforma o processador ARM em um CORE 16 bits, aumentando o espaço disponível na memória de programa. Essa mudança pode ser feita em tempo de execução, dessa forma o ARM7 possui dois conjuntos de instruções um de 32 bits e outro

de 16 bits. O segundo conjunto é mais simples que o primeiro, mas continua processando dados em 32 bits. Todos os registradores no modo THUMB continuam sendo acessados como 32 bits.

As instruções ARM fornecem diversas funcionalidades interessantes, uma delas é a execução condicional de todas as instruções. Os quatro últimos bits de cada instrução são comparados com os bits de estado do registrador CPSR, caso a comparação seja negativa a instrução é executada como um NOP. Dessa forma é minimizado o impacto das instruções de decisão no funcionamento do *pipeline*.

Podem-se realizar comparações sem precisar realizar saltos dependendo do resultado e assim evitando descarregar o conteúdo do *pipeline*.

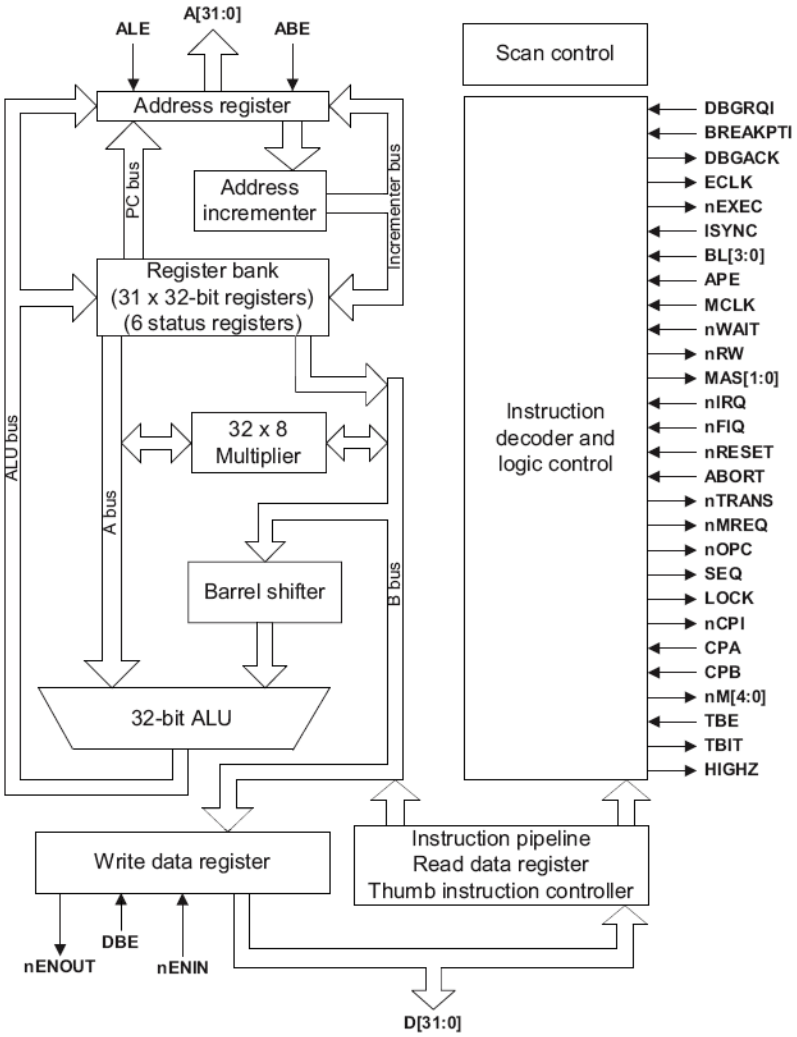
A *ARM Ltd* não produz os chips, mas vende licenças para sua manufaturação. Isto significa que existem muitas fontes e muitas versões diferentes apropriadas para quase qualquer finalidade. Para finalidades de controle o NXP LPC210x é provavelmente a melhor escolha. Ele tem o preço no mesmo nível de muitos microcontroladores de 8-bits anteriores e passa muitas vezes sua eficiência e performance.

O primeiro chip ARM foi projetado por Steve Furber e Sophie Wilson na *Acorn Computer* entre 1983-1985. Não existiam recursos disponíveis e sua simulação foi escrita em BASIC. O chip resultante tinha menos da metade do número de transistores do Motorola 68000 e batia-o de todas as formas, mesmo a divisão em software do ARM era mais rápida que na divisão em hardware do Motorola 68000.

Talvez os equipamentos eletrônicos mais comuns que usam este processador atualmente são:

- *iPod* da Apple
- *Nintendo Ds* e *Game Boy Advance* da Nintendo
- Maioria dos celulares Nokia
- Lego *Mindstorms NXT*
- Processamento de áudio no SEGA *Dreamcast*
- Receptores de rádio *Sirius Satellite*

2 Diagrama de Blocos da Arquitetura ARM7



3 Diferenças entre ARM7 e MIPS

O *pipeline*

O *pipeline* do ARM7 é de 3 estágios somente, sendo o do MIPS com 5 estágios. No ARM7 seus estágios são: busca da instrução; sua decodificação; execução da instrução. No MIPS existem esses três estágios mais o acesso a dados na memória e a escrita do resultado.

Acesso à memória

Para toda variável que for ser utilizada no processamento em um ARM, ela deve ser carregada antes porque todo processamento de dados é feito utilizando-se os registradores da CPU. No MIPS, diferentemente o acesso à memória está como estágio do processamento.

Modos de execução

O ARM possui o modo THUMB que transforma o processador em um CORE 16 bits, que pode ser ativado em tempo de execução, aumentando o espaço disponível na memória do programa. Mesmo com dois conjuntos de instrução (um de 32 bits e outro de 16 bits) o processamento continua sendo em 32 bits assim como o acesso aos seus registradores.

Registradores

A arquitetura ARM disponibiliza de 16 registradores. Qualquer dos registradores podem ser carregados ou gravados na memória com uma simples instrução.

Desvios condicionais

Os desvios condicionais levam em conta um registrador de status interno, ao contrário dos desvios condicionais do mips. que utilizam qualquer registrador para decidir sobre o desvio.

Desvios incondicionais

No arm os desvios incondicionais são executados movendo-se o endereço alvo para o registrador PC. No mips os desvios utilizam o comando “j $númeroDoRegistrador ”.

4 Implementações no ARM 7

As implementações de algoritmos de dump de registradores e multiplicação de inteiros foi feita utilizando o editor de texto Kate, compilada para ARM7 através de uma versão do gcc cross-compiler-arm-elf e testada no emulador gxemul.

A escolha pelo gxemul se deu pelo fato de ele possuir um bom sistema de depuração de código e dispositivos de saída de teste, o que permite a impressão dos resultados dos programa na tela sem muita dificuldade.

4.1 Algoritmo de dump de registradores

Arquivo: dump.asm (código Assembly compilável no gcc-cross-arm-elf)

```
.text
.globl dumpDeReg
dumpDeReg:
        # liberando espaço para 4 registradores na pilha
        add sp, sp, #-16
        # guardando o valor de r0
        # ele vai conter o valor do stack pointer corrigido, precisa ser salvo
        str r0, [sp, #12]
        # fixando r0 com o valor de sp
        mov r0, sp
       # corrigindo o valor do sp em r0 (foi diminuido de 16 para armazenar 4 registradores)
       # agora está sendo somado 16 para voltar a seu valor original
        add r0, r0, #16

     # Armazenando registradores em ordem decrescente para facilitar a implementação
    # da parte feita em C, que mostra os registradores
   # pc = r15, r14 = r15, r0 = r13 = sp (contém o valor corrigido de sp)
      str pc, [sp, #8]
      str r14, [sp, #4]
```

```
        str r0, [sp, #0]
 # recarregando o valor original de r0, para armazená-lo na pilha e mostrá-lo
        ldr r0, [sp,#12]
# armazenando registradores de r0 até r12 de maneira decrescente
        stmfd sp!, {R0-R12}
# fixando r0 como ponteiro para topo da pilha
        mov r0, sp
# chamando função que imprime os registradores
        bl mostrarRegs

# recarregando o valor original de sp
        ldr sp, [sp, #52]
# recarregando o valor original do registrador r14 no pc (r15)
# equivale ao "jr $ra" do mips
        ldr r15, [sp, #-12]
```

principal.c (Arquivo C compilável em gcc)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main() {
        dumpDeReg();
        halt();
}
void mostrarReg(int n, unsigned int *enderecoDoConteudo);
void mostrarRegs(unsigned int *enderecoBase) {
       int i;
       for (i = 0; i<16; i++, enderecoBase++)
              mostrarReg(i, enderecoBase);
}
```

```
void mostrarReg(int n, unsigned int *enderecoDoConteudo) {
        printstr("$");
        print_int(n);
        printstr(" = ");
        print_hex(enderecoDoConteudo);
        printstr(" -> ");
        print_hex(*enderecoDoConteudo);
        printstr("\n");
}
```

4.2 - Algoritmo de multiplicação de inteiros

Arquivo: multiplicar.asm (Compilável em gcc-cross-arm-elf)

```
#####################
.text
.globl multiplicar
multiplicar:
      # salvando fp, ip, lr e pc para recarregá-los ao fim da função
      stmfd   sp!, {fp, ip, lr, pc}

     #início da multiplicação
     # r0 = multiplicando, r1 = multiplicador
     # movendo o valor de r0 deslocado de 16 bits a esquerda para o registrador r2
      mov r2, r0, lsl #16
    # fixando r4 com o valor de r2
      mov r4, r2
   # executando função lógica OR, r1 OR r2 e guardando em r2
      orr r2, r2, r1
  # fixando r5 com o valor = 0, r5 será o controlador do laço..
      mov r5, #0
      laco:
     # executando r2 and 1.. verificando o valor do dígito menos significativo de r2
      and r3, r2, #1
     # deslocando r2 1 bit a esquerda (preparação para próximo laço)
```

```
    mov r2, r2, lsr #1
  # se o valor do último bit de r2 for maior que 0, adicionar o valor de r2 a r0
    cmp r3, #0
    bleq pular
    add r0, r0, r4
    pular:
 # deslocando r0 1 bit a direita
    mov r0, r0, lsr #1
 # adicionando 1 ao registrador que controla o laço
    add r5, r5, #1
# se o registrador de controle de laço for menor que 16, continuar laço
    cmp r5, #16
    blt laco
   # recarregando sp, fp. Carregando pc com antigo lr, mesmo que "jr $ra" do MIPS
    ldmfd   sp, {fp, sp, pc}
```