UNIVERSIDADE DE SÃO PAULO
Escola de Engenharia de São Carlos

Implementação de um core compatível com o MSP430 para FPGA

Juliano Alberto Paulino

São Carlos – SP

Juliano Alberto Paulino

IMPLEMENTAÇÃO DE UM CORE COMPATÍVEL COM O MSP430 PARA FPGA

Trabalho de Conclusão de Curso apresentado à Escola de Engenharia de São Carlos, da Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em Eletrônica

ORIENTADOR: Maximiliam Luppe

São Carlos

2012

Ficha catalográfica preparada pela Seção de Atendimentos ao Usuário do Serviço de Biblioteca – EESC/USP.

Paulino, Juliano Alberto
P328i Implementação de um core compatível com o MSP430 para FPGA. / Juliano Alberto Paulino; orientador Maximiliam Luppe. São Carlos, 2012.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2012.

1. FPGA. 2. MSP430. 3. Microcontroladores embarcados. 4. Desenvolvimento de hardware. 5. Soft-cores. I. Título.

FOLHA DE APROVAÇÃO

Nome: Juliano Alberto Paulino

Título: "Implementação de um Core Compatível com o MSP430 para FPGA"

Trabalho de Conclusão de Curso defendido e aprovado em 20 / 06 / 2012,

com NOTA 9,0 (nove, zero), pela comissão julgadora:

Prof. Dr. Maximiliam Luppe (Orientador) - EESC/USP

Prof. Associado Evandro Luís Linhari Rodrigues - EESC/USP

Profa. Assistente Luiza Maria Romeiro Coda - EESC/USP

Prof. Associado Homero Schiabel
Coordenador da CoC-Engenharia Elétrica
EESC/USP

Dedicatória

Dedico este trabalho à minha mãe Sonia e ao meu pai João que sempre me deram todo apoio e carinho durante toda minha vida além de todo o suporte necessário durante o período de minha graduação.

Agradecimentos

Agradeço primeiramente a Deus por todas as graças que proporcionou em minha vida.

À minha família, em especial à minha irmã Joyce e ao meu tio José Roberto, por todo apoio e incentivo nos momentos mais difíceis.

Ao meu professor e orientador, Maximiliam Luppe, por todos os ensinamentos e incentivo que proporcionaram a realização deste trabalho.

Ao meu amigo Guilherme pela valiosa amizade e aos meus amigos Bruno, Cleiton, Job e Stenio pelo companheirismo e por terem compartilhado comigo os momentos mais difíceis e também os mais divertidos da graduação.

Resumo

Este trabalho trata da implementação de um núcleo compatível com o microcontrolador MSP430 da Texas Instruments em FPGA utilizando a linguagem de descrição de *hardware* Verilog a fim de unir toda a popularidade e praticidade deste microcontrolador às vantagens dos microcontroladores embarcados em FPGA. Ao longo deste trabalho são discutidas características deste microcontrolador e são mostrados os detalhes de implementação da CPU e de alguns dos seus periféricos. Em seguida são mostrados resultados de diversas simulações feitas com os circuitos implementados.

Palavras-chave: FPGA, MSP430, microcontroladores embarcados, desenvolvimento de hardware, soft-cores.

Abstract

This paper deals with the implementation of a Texas Instruments MSP430 microcontroller compatible core for FPGA using Verilog hardware description language to join all the popularity and practicality of this microcontroller to the advantages of FPGA embedded microcontrollers. Along this paper, characteristics of this microcontroller will be discussed and details of the implementation of the CPU and some peripherals will be showed. Then various results of simulations made with the implemented circuits will be showed.

Keywords: FPGA, MSP430, embedded microcontrollers, hardware development, soft-cores.

Sumário

Lista de Figuras

Lista de Tabelas

1. INTRODUÇÃO

1.1. Contextualização

A utilização de microcontroladores embarcados em FPGAs possui muitas vantagens. A possibilidade de personalização do *hardware*, a redução de custos com componentes eletrônicos e a possibilidade de migrar entre diferentes arquiteturas de microcontroladores sem que haja necessidade de modificar o *hardware* externo à FPGA são alguns dos benefícios que tornam a utilização de *soft-cores* cada vez mais populares.

A flexibilidade que as FPGAs oferecem ao desenvolvedor de *hardware* permite a ele a escolha de quais periféricos usará em sua aplicação, sem deixa-lo preso às limitações dos microcontroladores convencionais. Além disso, permite a criação de periféricos totalmente personalizados de forma a aprimorar seu projeto, tornando-as uma alternativa excelente aos desenvolvedores que tenham requisitos específicos, fora dos padrões disponíveis no mercado.

A utilização de microcontroladores embarcados em FPGAs, muitas vezes, traz benefícios econômicos. Como a maioria dos circuitos digitais pode ser implementada na própria FPGA, o número de componentes eletrônicos, na maioria das vezes, é reduzido. Custos com decodificadores de *display*, circuitos de *debounce*, controladores de memória e muitos outros são reduzidos e, em alguns casos, até anulados. Além de diminuir o custo do projeto, em muitos casos, ocorre diminuição da área ocupada pelo circuito, outra qualidade muito desejada.

Outra grande vantagem na utilização de *soft-cores* ao invés de microcontroladores convencionais é a possibilidade de migração entre diferentes arquiteturas sem a necessidade de se modificar o *hardware*. Atualmente, existe uma ampla gama de microcontroladores embarcados em FPGA que permite ao desenvolvedor de *hardware* desenvolver seu projeto em diferentes plataformas, testá-las e escolher qual satisfaz melhor suas necessidades e posteriormente pode optar por trocá-la sem a necessidade de refazer as placas de circuito impresso.

Diante de todas essas vantagens, surgiu a ideia de implementar um *soft-core* que fosse compatível com o microcontrolador MSP430.

O MSP430 é um microcontrolador de 16 bits de arquitetura RISC desenvolvido pela Texas Instruments na década de 1990. Seu grande diferencial em relação aos concorrentes de mercado é o consumo extremamente baixo de energia. Além deste diferencial, o MSP430 possui muitas outras características positivas, como por exemplo, sistema de *clock* flexível, conjunto de instruções simples, fácil aprendizado, ampla variedade de periféricos e diversos modelos

disponíveis no mercado para as mais diversas aplicações. Todas essas qualidades e os mais de dez anos em que está disponível no mercado fizeram do MSP430 um microcontrolador muito popular, usado amplamente nos mais diversos projetos.

Sua característica de baixo consumo de potência faz com que seja um dispositivo muito popular em projetos de sistemas embarcados, uma área onde os fabricantes de FPGA têm se esforçado para entrar. Este ponto comum entre as duas tecnologias sugere que seria interessante aliar os recursos e a popularidade do microcontrolador MSP430 a todas as vantagens dos microcontroladores embarcados em FPGA.

Existem dois projetos de núcleos compatíveis com o microcontrolador MSP430 conhecidos: os projetos "open MSP430" e "Synthesizable MSP430". O projeto "open MSP430", atualmente, é composto pela implementação da CPU e de alguns periféricos do microcontrolador. O projeto "Synthesizable MSP430" constitui em uma CPU compatível com o MSP430 e encontra-se incompleto e aparentemente foi descontinuado.

A ideia inicial deste trabalho era dar continuidade ao projeto "Synthesizable MSP430", realizando a implementação dos circuitos que faltavam, porém foi constatado que não seria viável dar continuidade a ele. Então uma nova CPU e novos periféricos foram implementados.

1.2. Objetivos do Projeto

Este trabalho tem como objetivos empregar e aprimorar os conhecimentos obtidos durante o curso de graduação em Engenharia Elétrica com Ênfase em Eletrônica, especialmente nas áreas de sistemas digitais, arquitetura de computadores e linguagens de descrição de *hardware*, para promover a implementação em linguagem Verilog de um *soft-core* compatível com o microcontrolador MSP430.

Para tal finalidade, foram implementados os seguintes módulos:

- CPU de 16 bits com suporte a interrupções, capaz de executar todo o conjunto de instruções do MSP430;
- Módulo básico de clock;
- Watchdog timer;
- Timer A e
- Port digital.

Todos os módulos citados acima foram implementados seguindo as especificações da família MSP430x2xx. Alguns desses módulos possuem características analógicas que não foram implementadas devido ao caráter digital das FPGAs.

1.3. Organização da Monografia

A presente monografia está organizada em cinco capítulos da maneira descrita abaixo.

No capítulo 2 é apresentada uma revisão teórica sobre os principais conceitos dos assuntos envolvidos neste trabalho: o microcontrolador MSP430 e FPGAs.

No capítulo 3 são tratados os detalhes de implementação de cada um dos módulos do projeto.

O capítulo 4 traz os resultados dos testes e simulações realizadas com cada um dos módulos implementados e respectivas discussões pertinentes.

O capítulo 5 trata das conclusões gerais do trabalho.

2. REVISÃO TEÓRICA

Neste capítulo é feita uma revisão teórica abordando os assuntos mais relevantes deste trabalho. Primeiramente, são abordadas algumas características do hardware do microcontrolador MSP430. Em seguida, é feita uma revisão sobre FPGAs.

2.1. Os microcontroladores MSP430

A família MSP430 é uma família de microcontroladores de propósito geral de baixo consumo de potência desenvolvida pela Texas Instruments na década de 1990. É composta por microcontroladores de 16 bits de arquitetura Von Neumann. Seu conjunto de instruções é formado por 27 instruções físicas e 24 instruções emuladas, totalizando 51 instruções, que são discutidas na seção 2.12.

Devido às características de baixíssimo consumo de energia, alto desempenho e baixo custo, os microcontroladores MSP430 tornaram-se extremamente populares. Segundo a Texas Instruments, atualmente a família conta com cerca de 230 dispositivos diferentes, voltados tanto para aplicações gerais como específicas.

Os microcontroladores da família MSP430 podem ser identificados como mostrado na figura 1.

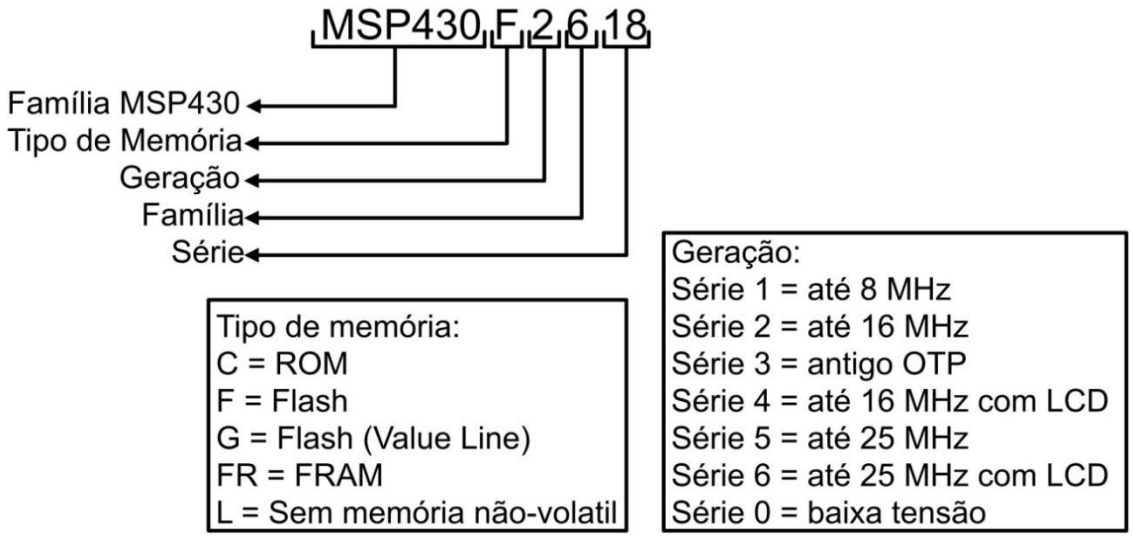

Figura 1 - Identificação dos microcontroladores MSP430.

Nas seções seguintes, serão mostradas algumas características dos microcontroladores MSP430.

2.1.1. A CPU

Os microcontroladores MSP430 possuem uma CPU de 16 bits de arquitetura RISC capaz de executar 27 instruções, a maioria delas em dois formatos diferentes, *byte* ou *word*, em até oito modos de endereçamento diferentes.

A CPU possui 16 registradores de 16 bits:

- Um contador de programa (R0/PC);
- Um ponteiro de pilha (R1/SP);
- Um registrador de status que também pode ser usado como gerador de constantes (R2/SR/CG1);
- Um gerador de constantes (R3/CG2) e
- Doze registradores de propósito geral (R4 a R15).

Dentre os registradores mencionados acima, dois deles merecem atenção especial: o registrador de status (R2/SR) e o gerador de constantes (R3/CG2).

O registrador R2/SR é responsável por armazenar as *flags* da CPU. Sua estrutura é mostrada na figura 2.

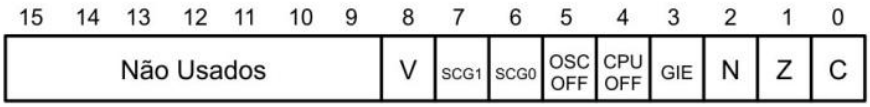

Figura 2 - Registrador R2/SR.

- O bit R2.0 é o *flag* de *carry*. Ele assume o valor 1 sempre que uma operação na ULA gera um *carry* de saída;
- O bit R2.1 é o *flag* de zero. Ele assume valor 1 sempre que o resultado de uma operação na ULA é igual a zero;
- O bit R2.2 é o *flag* de negativo. Ele assume valor 1 sempre que uma operação na ULA resulta em um número negativo;
- O bit R2.3 é o *Global Interrupt Enable*. Ele habilita todas as interrupções quando igual a 1 e desabilita todas as interrupções quando igual a 0;
- Os bits R2.4 e R2.5 desligam a CPU e o oscilador respectivamente quando iguais a 1;
- Os bits R2.6 e R2.7 são bits de controle do módulo básico de *clock* e
- O bit R2.8 é o *flag* de *overflow*. Ele é setado sempre que uma operação na ULA resulta em *overflow*.

Outra função do registrador R2, juntamente com o registrador R3, é a de gerador de constantes. Quando esses registradores são usados em alguma instrução como de operando de origem em conjunto com os diferentes modos de endereçamento, eles produzem constantes específicas, como mostradas na tabela 1.

Tabela 1 - Gerador de Constantes

Registrador origem (Rs)	Modo de endereçamento do registrador origem (As)	Constante gerada
R2	00	-
R2	01	(0)
R2	10	00004h
R2	11	00008h
R3	00	00000h
R3	01	00001h
R3	10	00002h
R3	11	0FFFFh

2.1.2. O conjunto de instruções

O conjunto de instruções do MSP430 é composto por 27 instruções físicas e mais 24 instruções emuladas, totalizando 51 instruções.

As instruções físicas são as instruções convencionais executadas pela CPU que possuem *opcodes* próprios. As instruções emuladas são aquelas que tornam o código-fonte mais simples de ser lido e escrito, mas não possuem opcodes próprios. Elas são substituídas automaticamente pelo *assembler* por instruções físicas (TEXAS, 2011).

Todas as instruções possuem 16 bits e podem ocupar uma, duas ou até três palavras na memória de programa, dependendo do modo de endereçamento utilizado.

Elas podem ser classificadas em três formatos diferentes:

- Instruções de um operando;
- Instruções de dois operandos e
- Instruções de salto.

As instruções de um operando possuem o formato ilustrado na figura 3.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Conteúdo	0	0	0	1	0	0	Opcode			B/W	As		Registrador Rs			

Figura 3 - Formato das instruções de um operando.

O conjunto das instruções de um operando e seus respectivos *opcodes* são mostrados na tabela 2.

Tabela 2 - Conjunto de instruções de um operando

Opcode			Mnemônico	Operação
0	0	0	RRC	Desloca um bit para a direita com *carry*
0	0	1	SWPB	Inverte a posição dos *bytes* mais e menos significativos
0	1	0	RRA	Desloca um *bit* para a direita aritmeticamente
0	1	1	SXT	Estende sinal
1	0	0	PUSH	Empilha
1	0	1	CALL	Chama sub-rotina
1	1	0	RETI	Retorna de interrupção

As instruções de dois operandos possuem o formato ilustrado na figura 4.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Conteúdo	Opcode				Reg. Origem (Rs)				Ad	B/W	As		Reg. Destino (Rd)			

Figura 4 - Formato das instruções de dois operandos.

O conjunto de instruções de dois operando e seus respectivos *opcodes* são mostrados na tabela 3.

Tabela 3 - Conjunto de instruções de um operando

Opcode				Mnemônico	Operação
0	1	0	0	MOV	Move a origem para o destino
0	1	0	1	ADD	Soma a origem ao destino
0	1	1	0	ADDC	Soma a origem e o *carry* ao destino
0	1	1	1	SUBC	Subtrai a origem do destino (com *carry*)
1	0	0	0	SUB	Subtrai a origem do destino
1	0	0	1	CMP	Compara a origem com o destino
1	0	1	0	DADD	Soma decimal da origem com o destino
1	0	1	1	BIT	Testa os bits da origem e do destino
1	1	0	0	BIC	*Reseta* o bit
1	1	0	1	BIS	Lógica Ou
1	1	1	0	XOR	Lógica Ou exclusivo
1	1	1	1	AND	Lógica E

As instruções de salto possuem o formato ilustrado na figura 5.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Conteúdo	0	0	1	Condição			Offset									

Figura 5 - Formato das instruções de salto.

O conjunto de instruções de salto e suas respectivas condições são mostradas na tabela 4.

Tabela 4 - Conjunto de instruções de salto

Condição			Mnemônico	Descrição
0	0	0	JNE/JNZ	Salta se o *flag* Z for igual a 0
0	0	1	JEQ/JZ	Salta se o *flag* Z for igual a 1
0	1	0	JNC/JLO	Salta se o *flag* C for igual a 0
0	1	1	JC/JHS	Salta se o *flag* C for igual a 1
1	0	0	JN	Salta se o *flag* N for igual a 1
1	0	1	JGE	Salta se N XOR V for igual a 0
1	1	0	JL	Salta se N XOR V for igual a 1
1	1	1	JMP	Salta incondicionalmente

As informações que cada instrução carrega são explicadas a seguir:

- **Opcode:** é o código de operação. É o conjunto de bits que especifica a operação a ser executada;
- **B/W:** é o bit que especifica o formato da instrução. B/W=0 para instruções do formato *word* e B/W=1 para instruções do formato *byte*;
- **As:** indica o modo de endereçamento do registrador origem. Os quatro possíveis modos de endereçamento para registradores origem são mostrados na tabela 5.

Tabela 5 - Modos de endereçamento do registrador origem

As		Modo de endereçamento	Descrição
0	0	Modo registrador	O operando encontra-se no registrador Rn
0	1	Modo indexado	O operando encontra-se no endereço de memória Rn + X
1	0	Modo indireto	O operando encontra-se no endereço de memória contido em Rn
1	1	Modo indireto/auto incremento	O operando encontra-se no endereço de memória contido em Rn. Em seguida Rn é incrementado em 2

- **Ad:** indica o modo de endereçamento do registrador destino (Rd). Ad=0 indica modo registrador, enquanto Ad=1 indica modo indexado.
- **Offset:** representa um número de 10 bits com sinal que é usado para calcular o novo valor do registrador PC caso a condição de salto seja satisfeita. Caso isso ocorra, o registrador PC é atualizado da seguinte forma:

$$PC_{novo} \leftarrow PC_{atual} + 2 \ + 2\,Offset$$

2.1.3 A organização da memória

O mapa da memória do MSP430 é mostrado na figura 6.

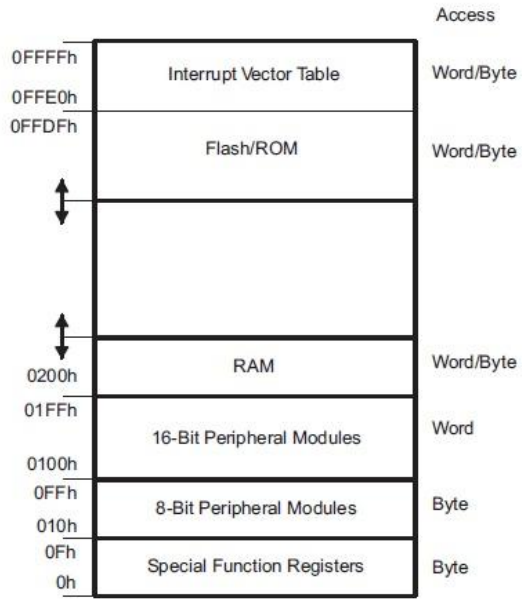

Figura 6 - Mapa da memória do MSP430. Figura retirada de (TEXAS, 2011).

A memória do MSP430 pode ser dividida em algumas regiões básicas:

- **Registradores de funções especiais:** localizados entre as regiões 00h e 0Fh. Esses registradores possuem como função habilitar funções em alguns módulos.
- **Registradores de periféricos:** localizados entre as regiões 0010h e 01FFh são os registradores responsáveis por fazer a comunicação entre CPU e periféricos. Alguns destes registradores possuem 8 bits e outros 16 bits.
- **RAM:** é a memória de dados utilizada para armazenar as varáveis do programa escrito pelo usuário. Inicia sempre no endereço 0200h e possui tamanho variável que depende do dispositivo.
- **ROM:** é a memória que armazena o programa escrito pelo usuário. O início da faixa depende do dispositivo. O fim é localizado no endereço 0FFFh.
- **Tabela do vetor de interrupções:** é uma região da memória ROM que armazena os endereços das rotinas de tratamento de interrupções. Está localizada entre os endereços 0FFE0h e 0FFFFh.

2.1.4. Os periféricos

Os microcontroladores da família MSP430 possuem uma gama muito ampla de periféricos. Ao longo dos anos, os dispositivos foram se aprimorando e novos periféricos foram desenvolvidos para suprir as necessidades específicas dos usuários do MSP430.

Dado o grande número de periféricos existentes, nesta seção serão tratados apenas os periféricos implementados no projeto:

- *Ports* digitais;
- Unidade básica de *clock*;
- *Watchdog timer* e
- *Timer A.*

2.1.4.1. Ports Digitais

Um microcontrolador interage de diversas formas com o sistema o qual faz parte. Ele pode receber entradas de seres humanos através de chaves, por exemplo. Pode exibir os resultados de suas operações através de *leds* ou *displays*. Pode receber sinais de sensores ou até mesmo se comunicar com outros microcontroladores.

O *port* é o periférico responsável pela comunicação do microcontrolador com o meio externo. Sua função é receber dados do meio externo e gravá-los em seu registrador de entrada e/ou escrever o conteúdo de seu registrador de saída nos pinos do microcontrolador.

Os *ports* do MSP430 possuem até 8 bits. Cada pino associado a um *port* pode ser configurado e controlado individualmente através dos registradores de controle do *port*. Cada *port* possui, ainda, registradores de entrada e saída individuais.

Os registradores associados aos *ports* digitais possuem todos 8 bits. São eles:

- **PXDIR**: é o registrador responsável por definir a direção do *port*. Cada bit deste registrador está associado a um pino do port. Bit=0 indica que o respectivo pino está selecionado como entrada. Bit=1 indica que o respectivo pino está selecionado como saída.
- **PXIN**: cada bit deste registrador assume o valor do sinal de entrada no pino correspondente quando este é configurado como entrada.
- **PXOUT**: cada bit deste registrador corresponde ao valor do sinal de saída no pino correspondente quando este é configurado como saída.

- **PXIFG**: cada bit deste registrador indica se há ou não interrupções pendentes. Bit=0 indica que não há interrupção pendente. Bit=1 indica que há interrupção pendente.
- **PXIE**: habilita (quando o bit=1) ou desabilita (bit=0) interrupção no pino correspondente.
- **PXIES**: cada bit deste registrador seleciona a borda de interrupção do pino correspondente. Bit=0 indica que o *flag* PXIFG passa do valor 0 para 1 na transição de nível lógico baixo para alto. Bit=1 indica que o *flag* PXIFG é passa do valor 0 para 1 na transição de nível lógico alto para baixo.

2.1.4.2. Módulo básico de clock

O módulo básico de *clock*, também conhecido como BCM, é o periférico responsável por gerar os sinais de *clock* que comandam o funcionamento dos componentes do microcontrolador.

Um diagrama simplificado desta unidade é mostrado na figura 7.

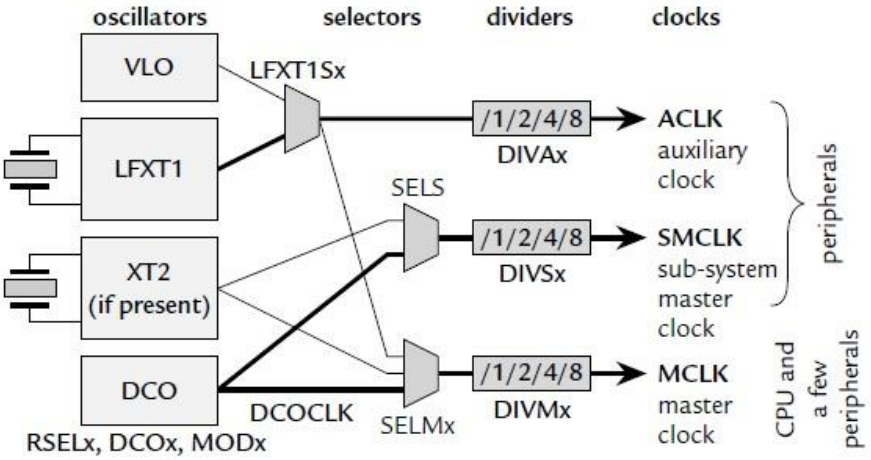

Figura 7 - Diagrama simplificado do módulo básico de *clock*. Figura retirada de (DAVIES, 2008).

Os sinais de *clock* podem ser obtidos de até quatro fontes diferentes descritas segundo (DAVIES, 2008) da seguinte forma:

- **VLO (*very low-frequency oscillator*)**: presente em poucos dispositivos, ele é um oscilador analógico interno de baixa frequência.

- **LFXT1 (*low - or high – frequency crystal oscillator*)**: presente em todos os dispositivos. É um sinal de *clock* externo geralmente ligado à um cristal de 32 kHz, mas que pode ser conectado à cristais de alta frequência de até alguns mega-hertz.
- **XT2 (*high-frequency crytal oscillator*)**: presente em poucos dispositivos, é similar ao LFXT1, exceto que é restrito a altas frequências.
- **DCO (*digital controlled oscilator*)**: presente em todos os dispositivos, ele é um dos grandes destaques do MSP430. É basicamente um oscilador RC altamente controlável.

A unidade básica de *clock* fornece como saída três sinais de *clock* que são distribuídos entre os componentes do microcontrolador. São eles:

- **MCLK (*Master clock*)**: é o sinal de *clock* usado pela CPU e por alguns periféricos.
- **SMCLK (*Subsystem master clock*)**: é o sinal de *clock* que é distribuído aos periféricos. Normalmente possui a mesma frequência do MCLK.
- **ACLK (*Auxiliary clock*)**: também é distribuído a alguns periféricos. Sua frequência é tipicamente 32 kHz.

A origem do sinal MCLK pode ser qualquer um dos sinais externos ou do oscilador interno DCO. O sinal SMCLK pode ter como origem o oscilador interno DCO ou algum sinal de baixa frequência dependendo do dispositivo. O sinal ACLK pode vir de qualquer uma das entradas de baixa frequência, dependendo também do dispositivo.

O módulo básico de *clock* é controlado por quatro registradores de 8 bits: DCOCTL, BCSCTL1, BCSCTL2 e BCSCTL3. A grande maioria das funções desses registradores é o controle de características analógicas dos osciladores internos, tais como a seleção de resistências e capacitâncias responsáveis pelo estabelecimento das frequências de operação.

2.1.4.3. Watchdog Timer

O *watchdog timer* é o periférico que possui a função de reiniciar o microcontrolador quando algum problema de software ocorre. Se a função *watchdog* não for necessária na aplicação, este periférico pode ser configurado como temporizador, gerando interrupções em intervalos de tempo regulares.

Sempre que o microcontrolador é reiniciado, o *watchdog timer* é configurado para funcionar no modo *watchdog* com intervalo de 32.768 ciclos de *clock*. O usuário deve desativar ou parar a contagem do *watchdog* antes que ela alcance seu valor máximo, caso contrário o microcontrolador será reiniciado.

A figura 8 mostra o diagrama de blocos do *watchdog timer*. Nesta figura são mostrados os principais componentes do *watchdog timer*, entre eles destacam-se o contador de 16 bits, também chamado de WDTCNT e o registrador de controle WDTCTL.

O contador WDTCNT é o componente principal do *watchdog timer*, através dele é feita a contagem dos pulsos de *clock* que determinam o funcionamento deste periférico. Seu valor não pode ser acessado por software. O contador WDTCNT pode ser configurado para contar até 64, 512, 8.192 ou 32.768 e pode ser excitado pelos sinais de *clock* ACLK ou SMCLK.

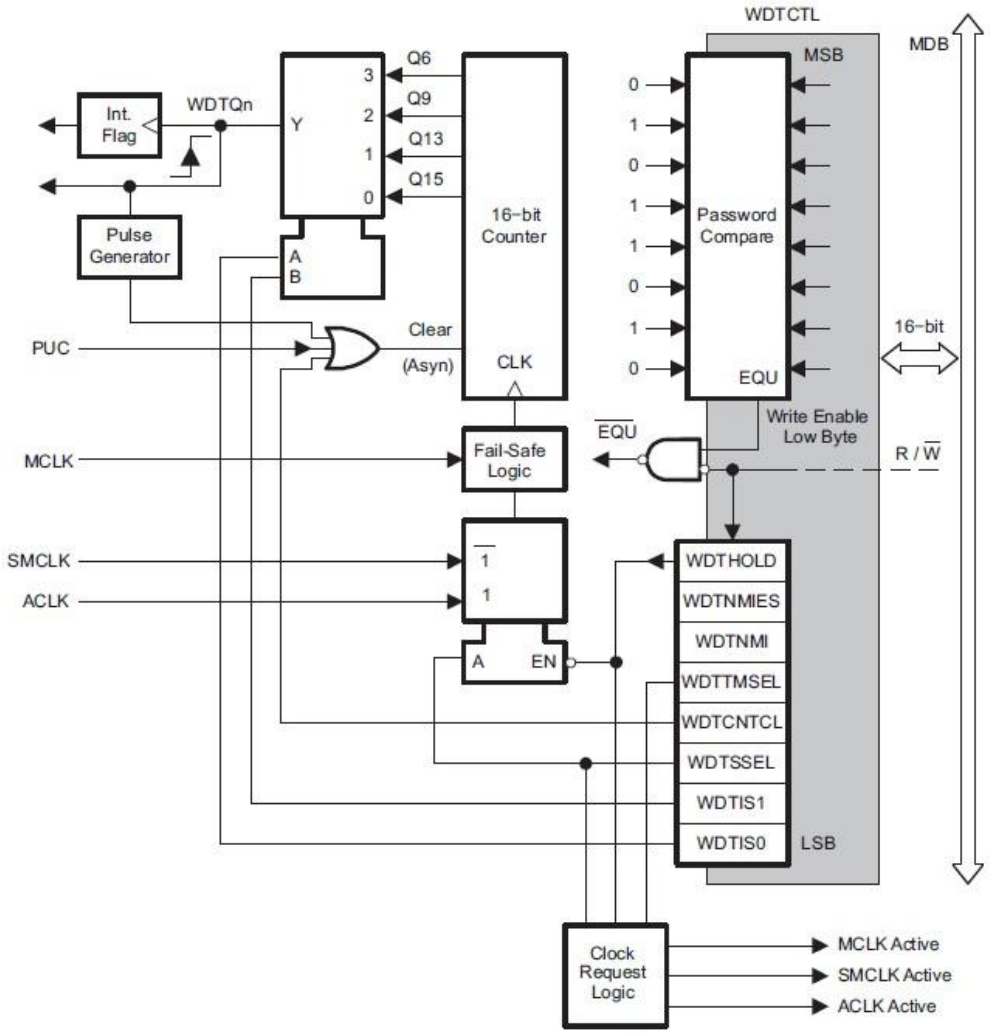

Figura 8 - Diagrama de blocos do *Watchdog Timer*. Figura retirada de (TEXAS, 2011).

O *watchdog timer* pode ser configurado através do registrador WDTCTL. Este registrador possui algumas particularidades. Ele é um registrador de 16 bits protegido por senha. Qualquer escrita neste registrador deve conter a senha 5Ah em seu *byte* mais significativo. Caso esta senha seja violada, o *watchdog timer* reinicia o microcontrolador imediatamente. Por outro lado, toda leitura do registrador WDTCTL retorna o valor 69h em seu *byte* mais significativo.

A figura 9 mostra a estrutura do registrador WDTCTL.

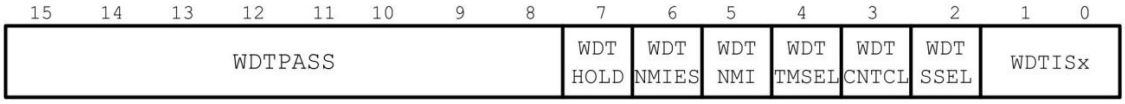

Figura 9 - Registrador WDTCTL.

Como pode ser observado na figura 9, o *byte* menos significativo do registrador WDTCTL contém os bits de controle do *watchdog timer*. São eles:

- **WDTHOLD:** habilita ou desabilita a contagem no contador WDTCNT.
- **WDTNMIES e WDTNMIE:** são usados para configurar o pino RST/NMI do microcontrolador.
- **WDTTMSEL:** seleciona o modo de operação, como *timer* ou *watchdog*.
- **WDTCNTCL:** é usada para reiniciar o valor do contador WDTCNT.
- **WDTSSEL:** seleciona a origem do sinal de *clock*, ACLK ou SMCLK, que excitará o contador WDTCNT.
- **WDTISx:** seleciona o intervalo de contagem de acordo com a tabela 6.

Tabela 6 - Intervalos de contagem do *watchdog timer*

WDTISx	Intervalo
00	Até 32.768
01	Até 8.192
10	Até 512
11	Até 64

2.1.4.4. Timer A

O *timer A* é o temporizador mais versátil, de propósito geral e presente em todos os dispositivos da família MSP430 (DAVIES, 2008). Seu *hardware* é dividido em dois blocos: o bloco temporizador e os canais de captura/comparação.

A figura 10 mostra o diagrama de blocos do *timer A*, onde é possível observar os dois blocos principais que o compõe.

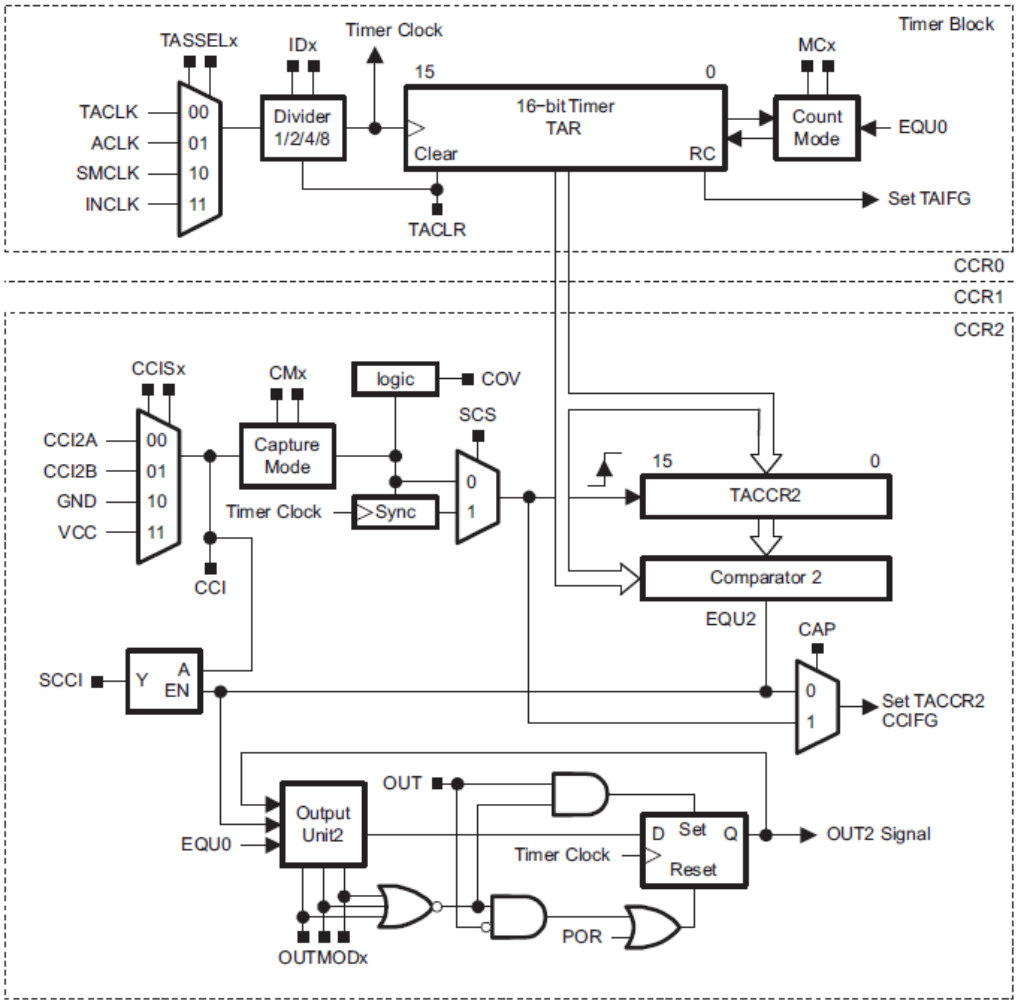

Figura 10 - Timer A. Figura retirada de (TEXAS, 2011).

O bloco temporizador tem seu funcionamento baseado no contador de 16 bits TAR. Esse contador pode ser excitado pelos sinais ACLK e SMCLK gerados pelo módulo básico de *clock* ou por sinais externos TACLK e INCLK. Cada um desses sinais pode ter sua frequência dividida em dois, quatro ou oito se necessário.

O contador TAR pode operar de três formas diferentes:

- Modo crescente: contando até o valor armazenado no registrador TACCR0 e reiniciando a contagem logo em seguida;
- Modo contínuo: contando até 0FFFFh e reiniciando a contagem quando ocorrer *overflow* do registrador TAR;
- Modo crescente/decrescente: contando de forma crescente até o valor armazenado em TACCR0 e então decrescente até 00000h, reiniciando o processo.

O bloco temporizador é controlado pelo registrador TACTL e possui uma *flag* de interrupção TAIFG que assume nível lógico alto sempre que a contagem no registrador TAR se torna igual a 0000h.

O bloco de captura/comparação tem seu funcionamento baseado no registrador TACCRx. Este bloco pode funcionar no modo captura, capturando o valor do registrador TAR quando uma de suas entradas for excitada ou como comparador, solicitando interrupção quando o valor do registrador TAR se torna igual ao valor armazenado em TACCRx.

O bloco de captura/comparação é controlado pelo registrador TACCTLx e possui um flag de interrupção CCIFG. Além da interrupção, este bloco pode fornecer como saída o sinal OUTx que pode ser configurado pelo usuário.

2.2. FPGAs

2.2.1. Arquitetura de FPGAs

FPGA (*field programmable gate array*) é um tipo de dispositivo lógico-programável que pode ser usado para a implementação de circuitos digitais complexos que usem até o equivalente a alguns milhões de portas lógicas (NAVABI, 2007). Sua principal vantagem é a capacidade de poder ser reconfigurada em campo, permitindo que sejam feitas correções de erros e adição de funções de *hardware*.

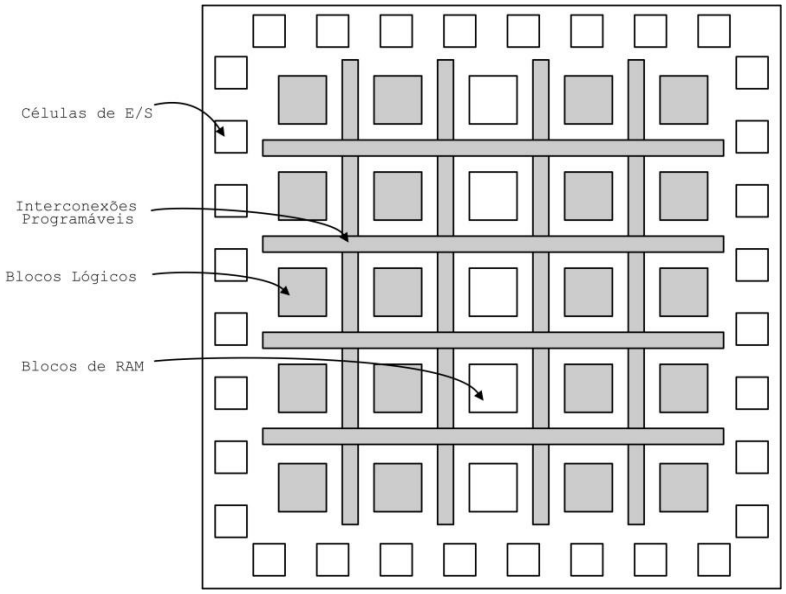

Figura 11 - Estrutura geral das FPGAs. Figura adaptada de (NAVABI, 2007).

Como pode ser visto através da figura 11, uma FPGA consiste em uma matriz de blocos lógicos (LABs) que podem ser conectados por redes de conexões programáveis horizontais e verticais. Além de blocos lógicos, existem células de E/S responsáveis pela interface do *hardware* implementado na FPGA com o meio externo e blocos de RAM que podem ser usados para a implementação lógica ou ser configurados como memória de diferentes tamanhos e endereços.

Os blocos lógicos consistem em uma matriz formada por elementos lógicos (LEs) interconectados localmente. A arquitetura dos blocos lógicos e o número de elementos lógicos em cada um desses blocos variam de acordo com o fabricante e a família a que o dispositivo pertence. A figura 12 mostra a arquitetura de um bloco lógico da família Cyclone II da Altera.

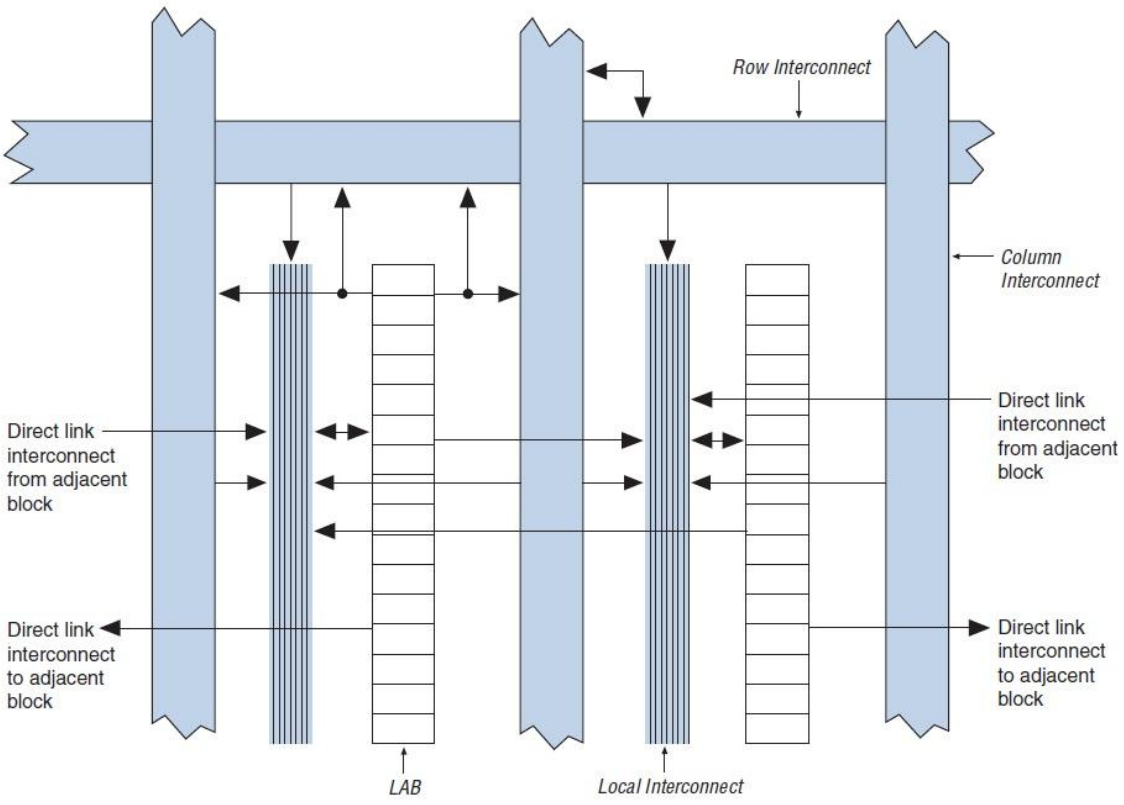

Figura 12 - Bloco lógico (LAB). Figura retirada de (ALTERA, 2008).

Os elementos lógicos, por sua vez, são a menor unidade lógica das FPGAs. Cada elemento lógico é composto, de forma simplificada, por uma tabela LUT (*look-up table*), um flip-flop e um multiplexador. A tabela LUT é definida por (TOCCI; WIDMER; MOSS, 2008) como a porção de um elemento lógico programável que gera uma função combinacional. Essa

função pode ser registrada (através do *flip-flop*) ou usada como saída do elemento lógico, a função é definida pelo multiplexador. A figura 13 mostra o diagrama de blocos de um elemento lógico da família Cyclone II da Altera.

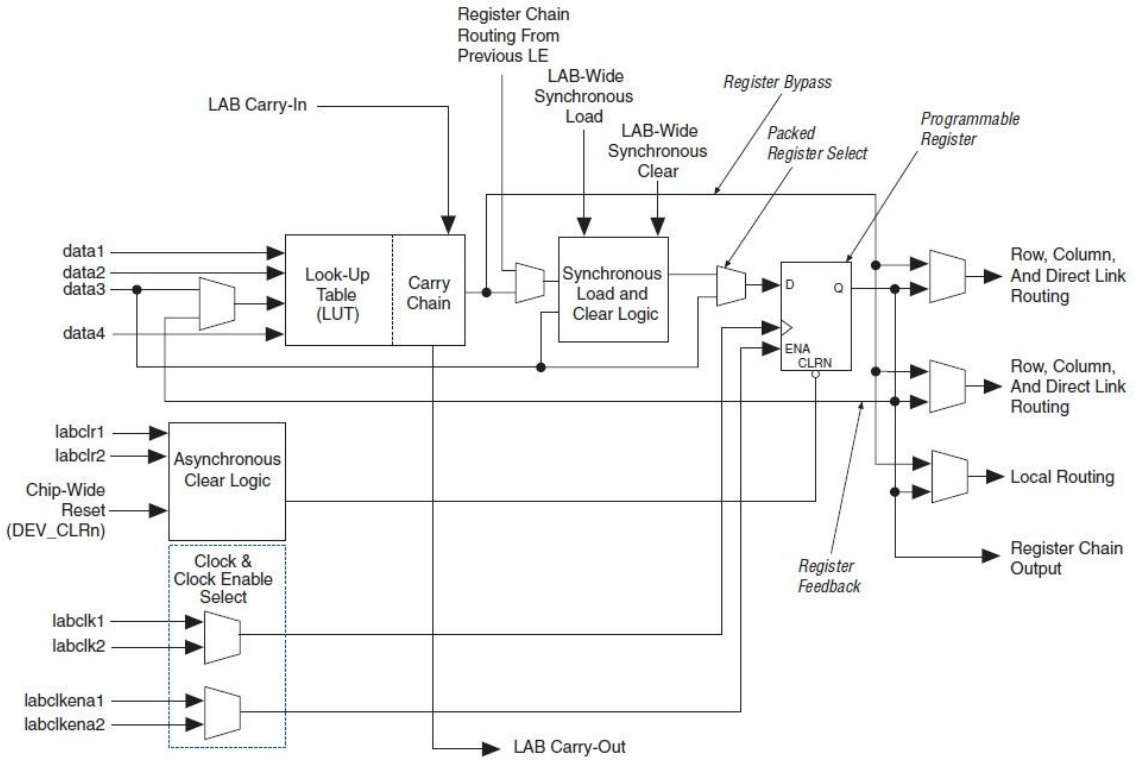

Figura 13 - Elemento lógico. Figura retirada de (ALTERA,2008).

2.2.2. Desenvolvimento em FPGAs

Segundo (BARR, 1999), o processo de criação de lógica digital não é muito diferente do processo de desenvolvimento de software embarcado. Uma descrição do comportamento e da estrutura de *hardware* é escrita numa linguagem de descrição de *hardware* de alto nível e esse código é então compilado e gravado no dispositivo. Uma alternativa às linguagens de descrição de *hardware* é a descrição do circuito através de esquemas de diagramas de blocos, porém este tipo de representação se torna menos popular quando os projetos são complexos.

Ainda segundo (BARR, 1999), a maior diferença entre desenvolvimento de *software* e *hardware* é a forma como o desenvolvedor deve pensar sobre o problema. Desenvolvedores de *software* tendem a pensar sequencialmente, mesmo em aplicações paralelas, pois as linhas de código-fonte que eles escrevem são executadas na ordem definida pelo programador. Já o

desenvolvedor de hardware deve pensar e programar de forma paralela, pois os sinais são processados de forma paralela enquanto caminham através dos mecanismos de um circuito.

O processo de desenvolvimento de circuitos lógicos possui cinco etapas:

1. Especificação da lógica;
2. Simulações;
3. Síntese;
4. *Place and route* e
5. Gravação.

A etapa de especificação da lógica corresponde à fase em que o circuito lógico é descrito através de diagramas de blocos esquemáticos ou de alguma linguagem de descrição de hardware, como Verilog ou VHDL.

Após a etapa de especificação da lógica, vem a etapa de simulações, onde *testbenches* são desenvolvidos com a finalidade de testar e validar a lógica implementada. O desenvolvimento de *testbenches* eficientes é uma tarefa difícil e trabalhosa, especialmente em sistemas grandes e complexos.

Após a validação da lógica implementada, vem a etapa de síntese. Nesta etapa, o código-fonte é compilado. O resultado desta etapa resulta numa representação da lógica implementada chamada *netlist*. O *netlist* independe do dispositivo em particular e é geralmente gravado num formato chamado *Electronic Design Interchange Format* (EDIF).

A etapa seguinte à síntese é a etapa chamada *place and route* (posicionar e rotear). Esta etapa consiste em mapear as estruturas lógicas descritas pelo *netlist* em interconexões, pinos de entrada e saída e blocos lógicos. O resultado do *place and route* é um *bitstream* que consiste nos dados binários que devem ser carregados na FPGA para que ela execute a função descrita pelo desenvolvedor.

Após a criação do *bitstream*, é necessário gravá-lo na FPGA. No mercado existem dispositivos com diferentes tipos de memória. Alguns desses dispositivos possuem memórias não temporárias como EEPROM e *flash*. Outros possuem memórias temporárias como SRAM, muito usadas em computação reconfigurável, que devem ser gravadas sempre que se for utilizar o dispositivo.

Para auxiliar no processo de desenvolvimento em FPGAs, os fabricantes fornecem ferramentas compatíveis com seus dispositivos. Por exemplo, a Altera possui o Quartus II, a Xilinx fornece o ISE Design Suite.

Neste projeto, foram usadas duas ferramentas: o Quartus II 11.1 Web Edition da Altera para a descrição dos circuitos e o ModelSim 10.0c Starter Edition criado pela Mentor Graphics para fazer as simulações do projeto.

3. MÉTODOS

Neste capítulo serão descritos os métodos utilizados na implementação dos circuitos que compõem o microcontrolador. A figura 14 mostra um diagrama de blocos indicando os componentes do sistema.

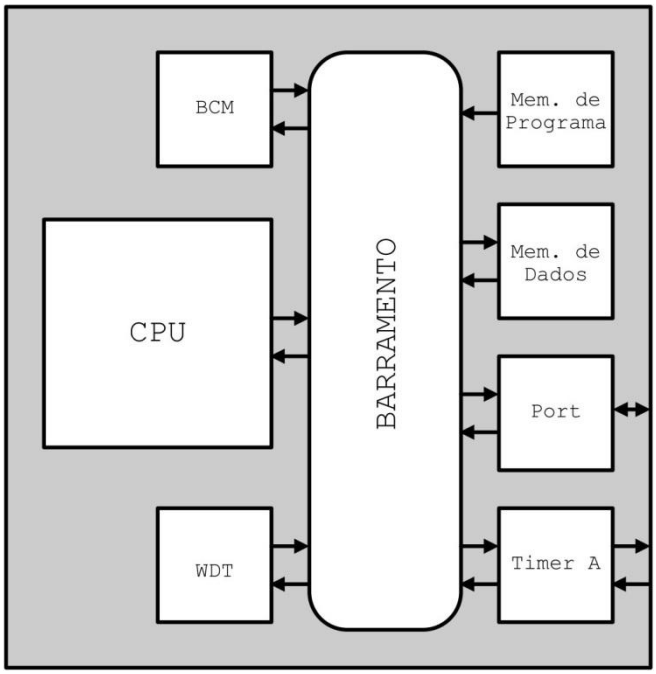

Figura 14 - Diagrama de blocos do microcontrolador implementado.

As seções seguintes discutirão em detalhes a implementação dos diversos componentes do microcontrolador.

3.1. CPU

Esta seção descreve em detalhes a implementação da CPU. Ela é, sem dúvida, o componente mais complexo de todo trabalho. Para simplificar o projeto da CPU, ela foi dividida em três módulos:

- A unidade de decodificação de instruções (IDU);
- O datapath e
- A unidade de controle (UC).

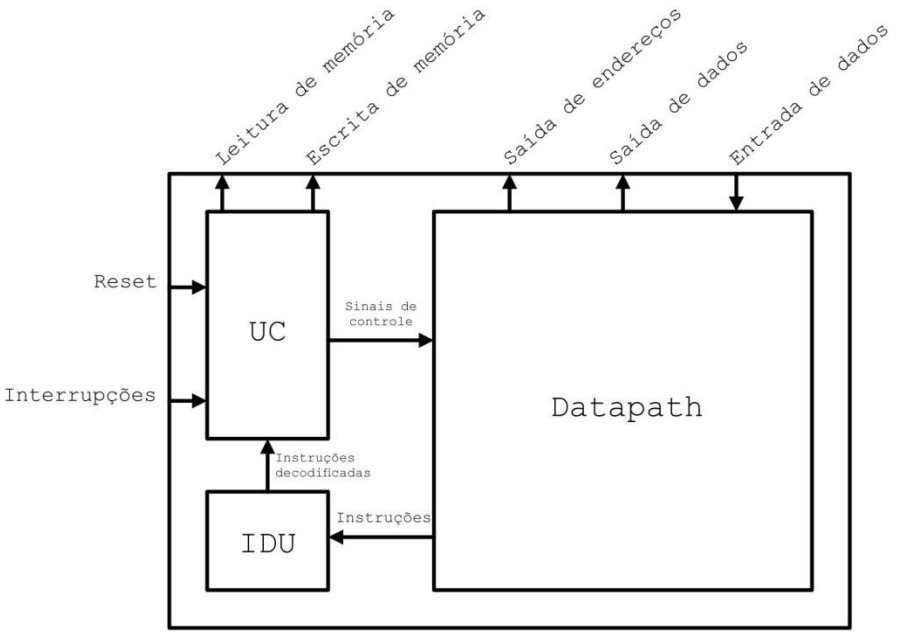

Figura 15- Diagrama da CPU.

A figura 15 ilustra um diagrama da CPU implementada e os três módulos que a compõem. Estes três módulos serão detalhados nas seções seguintes.

3.1.1. Unidade de decodificação de instruções

A unidade de decodificação de instruções é o módulo mais simples da CPU. Sua função é ler a instrução armazenada no registrador de instruções, identificar a qual das três categorias de instruções ela pertence e fornecer como saída todas as informações contidas nesta instrução de forma decodificada.

A instrução decodificada será usada, então, para alimentar a máquina de estados da unidade de controle. As informações contidas na instrução servirão para especificar a forma com que a operação será realizada pela CPU.

O diagrama mostrando os sinais de entrada e saída da unidade de decodificação de instruções é mostrado na figura 16.

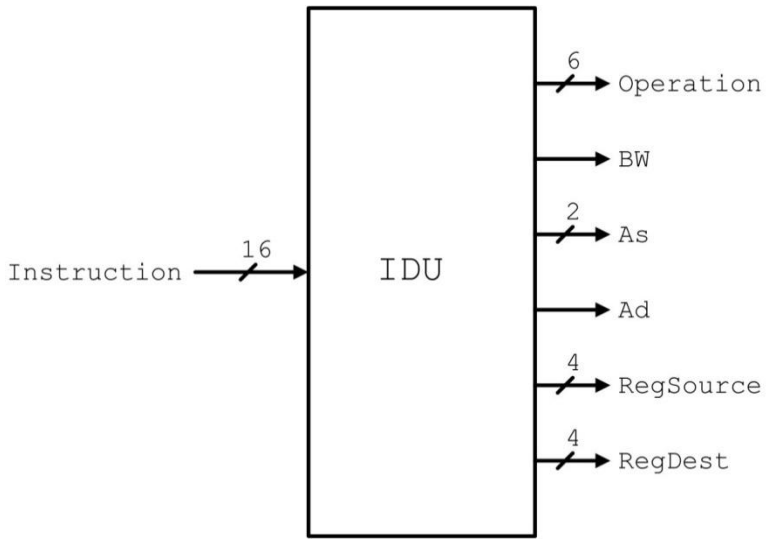

Figura 16 - Unidade de decodificação de instruções.

A descrição detalhada de cada um dos sinais da unidade de decodificação de instruções é mostrada na tabela 7.

Tabela 7 - Sinais da unidade de decodificação de instruções

Sinal	Tipo	Largura [bits]	Descrição
Instruction	Entrada	16	Instrução a ser decodificada.
Operation	Saída	6	Tipo de instrução e operação a ser executada pela ULA.
BW	Saída	1	Formato da operação (byte ou word).
As	Saída	2	Modo de endereçamento do registrador origem.
Ad	Saída	1	Modo de endereçamento do registrador destino.
RegSource	Saída	4	Registrador origem.
RegDest	Saída	4	Registrador destino.

A decodificação das instruções segue uma lógica simples. Conhecendo-se os formatos das instruções de um operando (Figura 3), de dois operandos (Figura 4) e de salto (Figura 5), o tipo de instrução pode ser identificado por meio dos bits mais significativos de cada tipo de instrução.

Observa-se que todas as instruções de um operando possuem a sequência 000100b em seus bits 6 bits mais significativos. Da mesma forma, as instruções de salto possuem a sequência 001b em seus 3 bits mais significativos. Essas características permitem identificar a qual das três classes de instruções, a instrução a ser decodificada pertence.

Depois de identificada, a decodificação da instrução torna-se simplesmente uma tarefa de associar os bits de entrada à sua respectiva saída.

A lógica empregada na unidade de decodificação de instruções, descrita aqui, pode ser representada de forma simples através do fluxograma da figura 17.

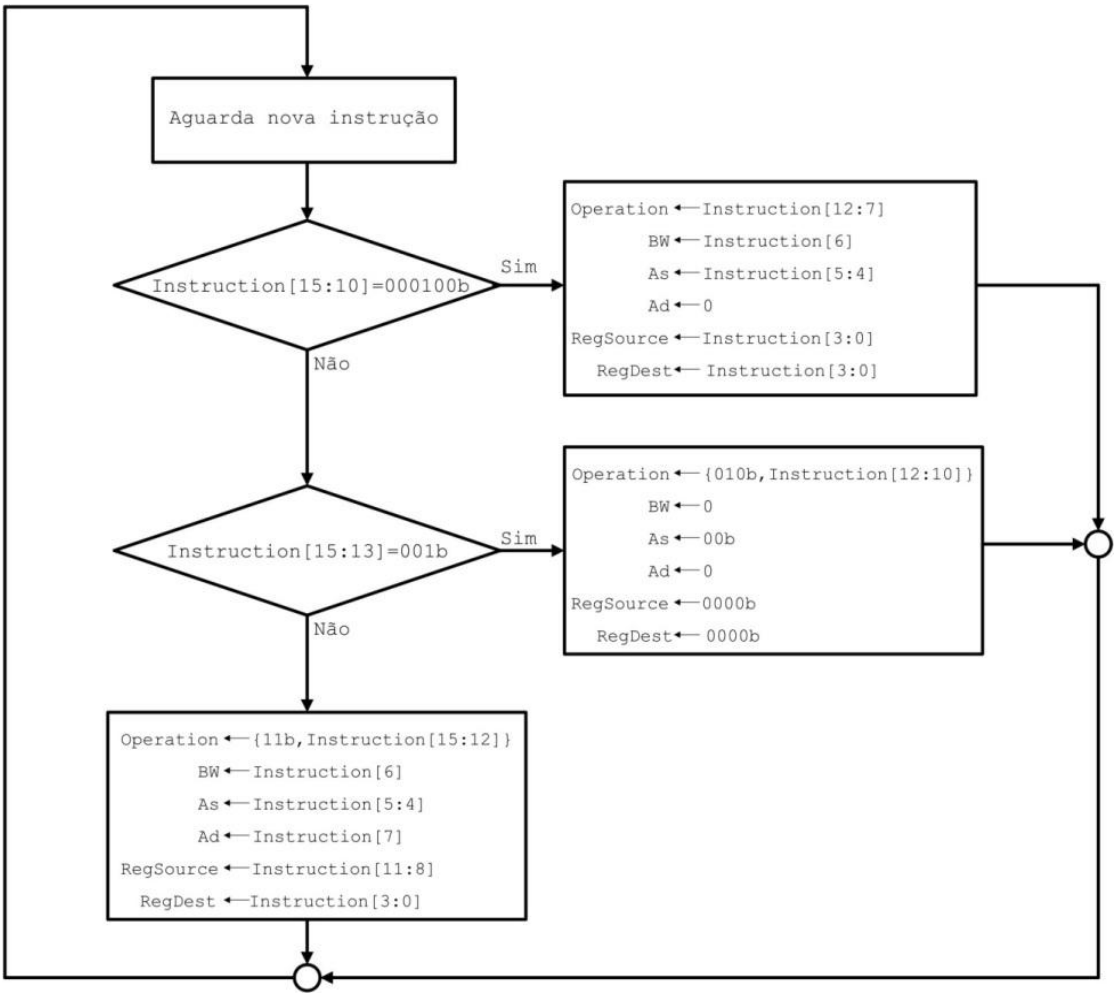

Figura 17 - Fluxograma da unidade de decodificação de instruções.

O fluxograma ilustrado na figura 17 ilustra exatamente a lógica empregada na descrição comportamental da unidade de decodificação de instruções. A cada atualização na instrução de entrada, a lógica da figura 16 é executada.

Vale acrescentar que o sinal de saída "*Operation*" é uma forma de representar os *opcodes* de maneira padronizada. Note, pelas figuras 3 e 4, que os opcodes das instruções de um e dois operandos possuem tamanhos diferentes, três e quatro bits respectivamente. Representar

todas as operações com o mesmo número de bits torna o projeto da unidade de controle e da ULA mais simples.

O formato padronizado em 6 bits para os opcodes é mostrado na tabela 8. Nesta tabela, as expressões XXX, YYY e ZZZZ representam os opcodes não padronizados das instruções de um operando, salto e dois operandos respectivamente.

Tabela 8 - Formato dos opcodes padronizados

Tipo de operação	Opcode padronizado					
	5	**4**	**3**	**2**	**1**	**0**
Operação de um operando	1	0	0	X	X	X
Salto	0	1	0	Y	Y	Y
Operação de dois operandos	1	1	Z	Z	Z	Z

Para finalizar a discussão sobre a unidade de decodificação de instruções, é importante mencionar que ela se trata de um circuito combinacional e a decodificação é instantânea (desconsiderando, é claro, o atraso do próprio *hardware*) e, portanto, não gasta nenhum ciclo de *clock* para ser realizada.

3.1.2. ULA

A unidade lógica e aritmética (ULA) é o circuito responsável por realizar todas as operações lógicas e matemáticas da CPU. O diagrama do circuito implementado é mostrado na figura 18.

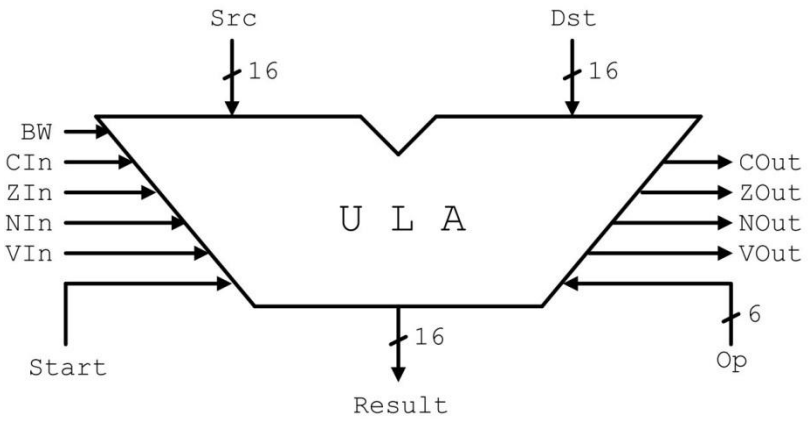

Figura 18 - Diagrama da unidade lógica e aritmética.

A descrição detalhada de cada um dos sinais de entrada e saída da unidade lógica e aritmética é mostrada na tabela 9.

Tabela 9 - Sinais da ULA

Sinal	Tipo	Largura[bits]	Descrição
Src	Entrada	16	Operando origem
Dst	Entrada	16	Operando destino
Op	Entrada	6	Código de operação
Start	Entrada	1	Realiza operação (em sua borda de subida)
CIn	Entrada	1	Entrada do flag Carry
ZIn	Entrada	1	Entrada do flag Zero
NIn	Entrada	1	Entrada do flag Negativo
VIn	Entrada	1	Entrada do flag de Overflow
BW	Entrada	1	Formato da operação (“byte” ou “word”)
Result	Saída	16	Resultado da operação
COut	Saída	1	Saída do flag Carry
ZOut	Saída	1	Saída do flag Zero
NOut	Saída	1	Saída do flag Negativo
VOut	Saída	1	Saída do flag Overflow

A ULA implementada neste projeto é capaz de realizar 16 operações que são mostradas na tabela 10. Dentre elas, doze são operações de dois operandos e quatro são operações de um único operando. A maioria delas pode ser executadas em dois formatos diferentes, *byte* ou *word*.

As operações no formato *word* utilizam operandos de 16 bits e fornecem resultados também de 16 bits. Já as operações no formato *byte* usam operandos de 8 bits, considerando apenas os 8 bits menos significativos de cada entrada de operando. O resultado das operações de formato *byte* são sempre apresentados nos 8 bits menos significativos da saída “Result”. Os 8 bits mais significativos são iguais a 0.

Embora 4 bits sejam suficientes para endereçar todas as 16 operações realizadas pela ULA, o código de operação utilizado possui 6 bits por conveniência. Desta forma o código de operação gerado pela unidade de decodificação de instruções (o código padronizado que possui 6 bits) pode ser usado diretamente na unidade lógica e aritmética sem a necessidade de adaptar esse código a um formato de 4 bits.

As operações são executadas pela ULA sempre na borda de transição positiva do sinal "Start". O resultado é mantido no registrador de saída até a próxima transição positiva deste mesmo sinal.

Tabela 10 - Operações executadas pela ULA

Código de operação (Op)	Instrução relacionada	Operação realizada
110100b	MOV	Result ← Src
110101b	ADD	Result ← Src + Dst
110110b	ADDC	Result ← Src + Dst + Cin
111000b	SUB	Result ← Dst - Src
110111b	SUBC	Result ← Dst – Src + Cin
111001b	CMP	Dst – Src, mas Result ← Dst
111010b	DADD	Result ← Src + Dst (Soma BCD)
111111b	AND	Result ← Src AND Dst
111011b	BIT	Src AND Dst, mas Result ← Dst
111100b	BIC	Result ← NOT(Src) E Dst
111101b	BIS	Result ← Src OR Dst
111110b	XOR	Result ← Src XOR Dst
100000b	RRC	Cin→MSB→MSB-1....LSB+1→LSB→COut
100010b	RRA	MSB→MSB→MSB-1→...LSB+1→LSB→COut
100001b	SWPB	Result[15:0] ← { Dst[7:0] , Dst[15:8] }
100011b	SXT	Result[15:8] ←Dst[7] e Result[7:0] ←Dst[7:0]

3.1.3. *Datapath*

O *datapath* é o circuito responsável por executar as operações da CPU. Ele é composto pelos seguintes componentes:

- Unidade lógica e aritmética;
- Registradores da CPU;
- Registradores auxiliares;
- Multiplexadores;
- Multiplicador por 2 e
- Somador.

O circuito implementado é mostrado na figura 19.

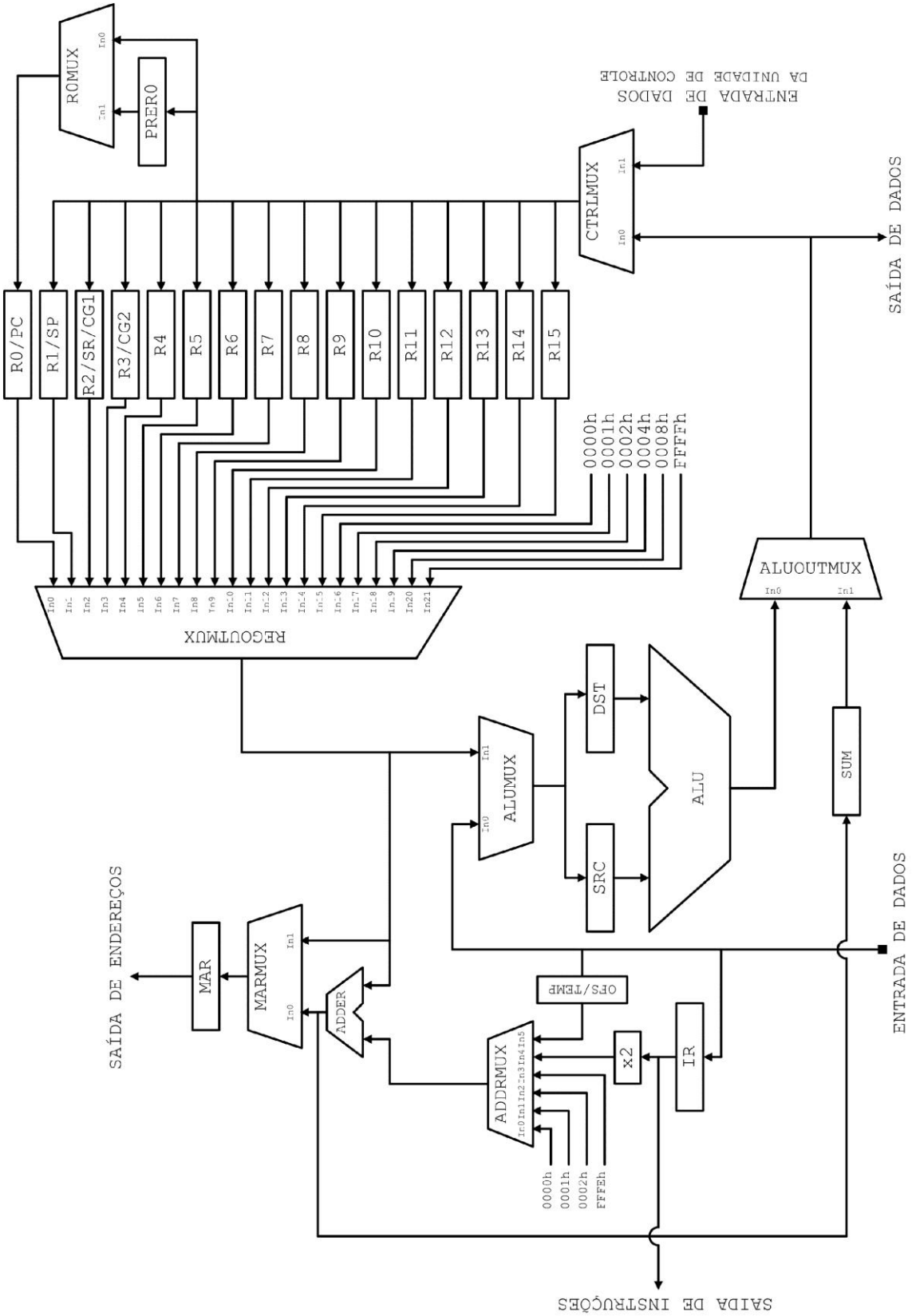

Figura 19 - Datapath adaptado de MSP430 Microarchitectural Simulator.

Como pode ser observado na figura 19, o *datapath* possui três saídas. A "Saída de Endereços" é usada para selecionar endereços da memória nas operações de escrita ou leitura. A função da "Saída de Dados" é levar o resultado das operações realizadas no *datapath* à memória de dados ou a algum registrador externo à CPU. A "Saída de Instruções" indicará sempre o conteúdo do registrador de instruções IR. Ela é conectada à unidade de decodificação de instruções.

O *datapath* possui duas entradas. A "Entrada de Dados" é usada para carregar os operandos que se encontram nas memórias de dados, de programa ou nos registradores externos à CPU. A "Entrada de Dados da Unidade de Controle" é conectada diretamente à unidade de controle e é usada em duas situações. Durante o ciclo de *reset* ou quando há alguma solicitação de interrupção, os registradores PC e SR precisam ser carregados com valores específicos. Essa operação é feita através desta entrada com o multiplexador CTRLMUX na posição 1.

Todos os 16 registradores da CPU (R0 a R15) estão contidos no *datapath*. Eles podem ser selecionados pela unidade de controle através do multiplexador REGOUTMUX. Note que, além dos registradores, seis valores constantes estão ligados às entradas deste multiplexador. Eles são as constantes que podem ser obtidas através do gerador de constantes. Por exemplo, quando uma instrução usar o registrador R3 como operando de origem em conjunto com o modo de endereçamento As=00, o valor selecionado pelo multiplexador REGOUTMUX será 0000h ao invés do valor contido em R3. Para mais detalhes do uso do gerador de constantes, consulte a tabela 1.

O registrador PRER0 e o multiplexador R0MUX são usados em conjunto em certas situações quando há necessidade de se incrementar o valor de R0/PC e gravar algum dado em algum dos outros 14 registradores no mesmo ciclo de *clock*.

Os registradores SRC e DST são responsáveis por armazenar os operandos da ULA. Os operandos são carregados nesses registradores através do multiplexador ALUMUX. Esse multiplexador, por sua vez, possui duas entradas, uma delas é ligada a entrada de dados e a outra ligada à saída do multiplexador de registradores/constantes.

O registrador de endereço de memória, MAR, é responsável por armazenar os endereços de memória acessados pela CPU em um determinado instante. Seu conteúdo é carregado através do multiplexador MARMUX, cujas entradas são ligadas ao multiplexador de registradores e ao somador. O somador, por sua vez, é usado nas operações que demandam a soma de um valor a um endereço. Por exemplo, durante ciclo de busca de instrução, na etapa em que o PC é incrementado, o somador é carregado com o registrador PC em uma de suas entradas e a

constante +2 é carregada na outra através do multiplexador de endereços ADDRMUX, produzindo assim o valor PC+2 que será gravado no registrador PC.

Outro exemplo da aplicação do somador é nas operações que se utilizam do modo de endereçamento indexado. Neste modo de endereçamento, o conteúdo de um registrador deve ser somado a um índice para produzir um endereço. Esse índice é carregado da memória de programa no registrador OFS/TEMP e colocado na entrada do somador através do multiplexador de endereços. Enquanto isso, a segunda entrada do somador é alimentada com o valor contido no registrador indicado na instrução, produzindo assim o endereço indexado Rn + X.

O resultado das operações realizadas pelo somador muitas vezes devem ser armazenadas temporariamente, para isso utiliza-se o registrador SUM. Por exemplo, considere as operações com modo de endereçamento indireto com auto incremento (@Rn+). Neste tipo de operação o conteúdo de um dado registrador Rn deve ser usado como endereço e, no ciclo seguinte, incrementado em 2. Para armazenar o resultado da soma Rn+2 temporariamente até o instante conveniente para que seja atualizado no registrador Rn, utiliza-se o registrador SUM.

O multiplexador de saída da ULA, ALUOUTMUX, é usado apenas para selecionar qual informação vai para a saída de memória e registradores da CPU, um resultado da ULA ou uma soma de endereços feita pelo somador.

Por fim, o componente identificado na figura 19 como X2 é um multiplicador por 2 com extensão de sinal. Ele é usado nas instruções de salto. Sua função é multiplicar os 10 bits de *offset* das instruções de salto por 2 e estender seu sinal para os bits mais significativos produzindo um *offset* no formato de um número binário de 16 bits com sinal, que é a forma adequada para ser usada como operando do somador.

Quando ocorre um salto no programa, o registrador PC é atualizado da seguinte forma:

$$PC_{novo} \leftarrow PC_{atual} + 2 \ + 2\,Offset$$

A parcela de soma $PC_{atual} + 2$ é feita naturalmente durante o ciclo de busca da instrução pelo do somador e a multiplicação $2\,Offset$ é feita pelo multiplicador X2.

3.1.4. Unidade de controle

A unidade de controle consiste basicamente em uma máquina de estados finitos, cuja função é controlar todos os componentes do *datapath* e o barramento, coordenando assim o funcionamento do microcontrolador.

A saída desta máquina de estados pode ser definida como o conjunto dos sinais que controlam os componentes do *datapath* mais os sinais de leitura e escrita na memória.

A entrada desta máquina de estados é o conjunto formado pelos sinais de instruções decodificadas pela IDU, os sinais de interrupções dos periféricos e o sinal de *reset* do sistema.

A figura 20 mostra de forma simplificada o diagrama de estados da unidade de controle, desenhado com o propósito de facilitar o entendimento de seu funcionamento.

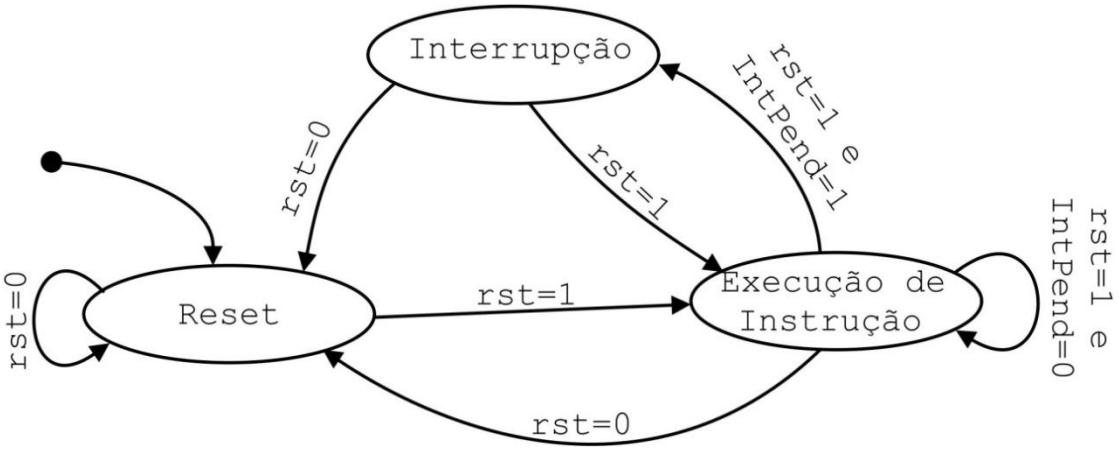

Figura 20 - Máquina de estados simplificada.

O funcionamento da CPU começa sempre no estado “Reset” após um pulso negativo do sinal de entrada “rst”. Esse estado é composto por dois sub-estados responsáveis por inicializar o registrador R0/PC e o registrador R2/SR com seus respectivos valores iniciais.

O estado de interrupção ocorre sempre que ao final da etapa de execução de uma instrução, há uma interrupção pendente sinalizada pelo registrador interno “IntPend”. Esse estado é composto por cinco sub-estados como mostrado na figura 21.

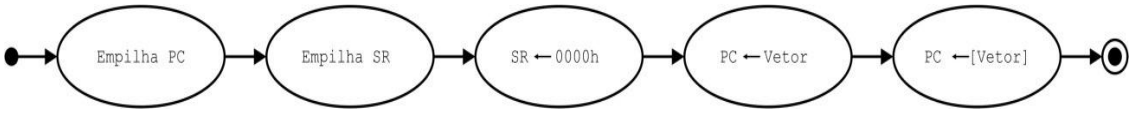

Figura 21 – Sub-estados de interrupção.

Quando a CPU entra no estado de interrupção, os registradores PC e SR são empilhados na memória de dados a partir do endereço indicado pelo ponteiro de pilha. Em seguida o registrador SR é carregado com o valor 0000h e o registrador PC apontará para o endereço do

vetor de interrupções correspondente à interrupção solicitada. O endereço armazenado no vetor será então carregado no registrador PC para que a execução do *interrupt handler* programado pelo usuário seja feita a partir do próximo ciclo de busca de instrução.

Na unidade de controle implementada, há suporte a quatro interrupções: interrupção do *Port* 1 (prioridade 18), interrupções do *Timer A* (prioridades 24 e 25) e interrupção do *watchdog timer* (prioridade 26). As prioridades seguem o *datasheet* do microcontrolador MSP430G2231. Caso duas interrupções ocorram simultaneamente, a de maior prioridade será executada primeiro.

O estado denominado "execução de instrução" na figura 20 é o estado mais complexo assumido pela unidade de controle. A forma com que as instruções são executadas depende de duas variáveis: o tipo de instrução e o modo de endereçamento. De forma geral, as instruções são executadas seguindo as etapas mostradas na figura 22, porém a forma como a unidade de controle atua no ciclo de busca do operando de origem, por exemplo, depende de seu modo de endereçamento (As), da mesma forma que os ciclos de busca de operando destino e gravação dependem do modo de endereçamento do operando de destino (Ad).

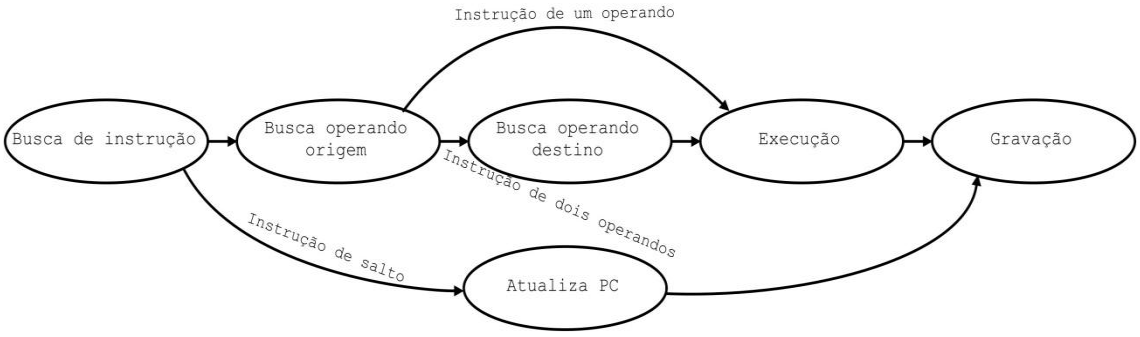

Figura 22 - Execução de instruções.

A seguir é apresentado um resumo, etapa por etapa, de como as instruções são executadas pela unidade de controle, dados o tipo de instrução e o modo de endereçamento de seus operandos. Cada tipo de instrução é representado por uma tabela contendo as ações de controle tomadas em cada etapa de execução e um diagrama de tempo mostrando como os ciclos de execução estão distribuídos ao longo do tempo.

Nestes diagramas, o ciclo de busca e decodificação de instruções compreendem o intervalo de tempo em que a instrução é carregada no registrador IR, decodificada e o registrador PC incrementado. Os ciclos de busca de operando de origem e destino são caracterizados pelas etapas em que a CPU carrega esses operandos de um determinado local da

memória nos registradores DST e SRC do *datapath*, compreendendo as etapas de busca de *offset*, cálculo do endereço (para instruções com endereçamento indexado) e incremento de registradores. O ciclo de execução é caracterizado pelo instante em que o sinal "AluStart" passa do nível lógico baixo para o nível lógico alto, fornecendo, na saída da ULA, o resultado da operação desejada. O ciclo de gravação, por sua vez, compreende as etapas em que o resultado da operação é gravado na memória e os registradores, quando necessário, são incrementados.

- **Instruções de um operando com modo de endereçamento As=00b**

Etapa 1	Etapa 2	Etapa 3
IR ← (PC)	PC ← PC + 2 DST ← Rs	AluStart ← 1 Rs ← Result

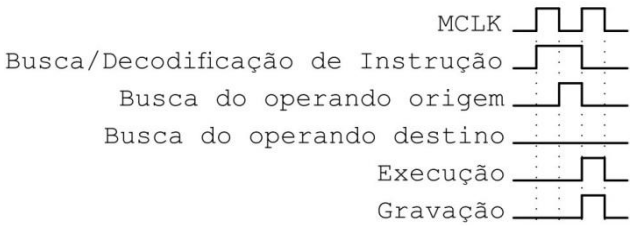

Figura 23 - Instruções de um operando com modo de endereçamento As=00b.

- **Instruções de um operando com modo de endereçamento As=01b**

Etapa 1	Etapa 2	Etapa 3	Etapa 4
IR ← (PC)	PC ← PC + 2 Ofs ← (PC)	PC ← PC + 2 Dst ← (Rs + Ofs)	AluStart ← 1 (Rs + Ofs) ← Result

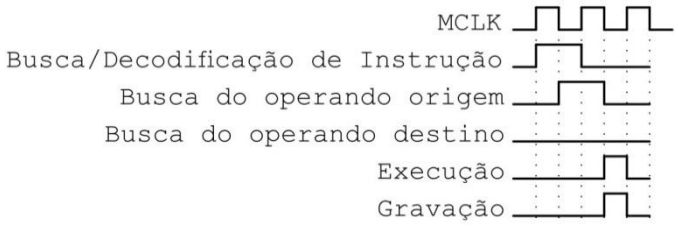

Figura 24 - Instruções de um operando com modo de endereçamento As=01b.

- **Instruções de um operando com modo de endereçamento As=10b**

Etapa 1	Etapa 2	Etapa 3
IR ← (PC)	PC ← PC + 2 Dst ← (Rs)	AluStart ← 1 (Rs) ← Result

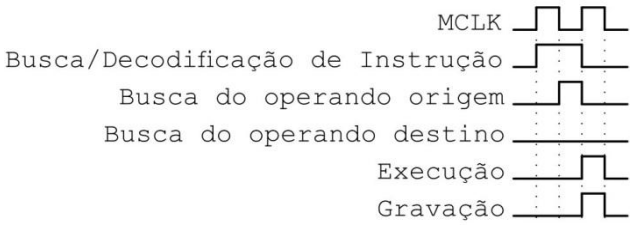

Figura 25 - Instruções de um operando com modo de endereçamento As=10b.

- **Instruções de um operando com modo de endereçamento As=11b**

Etapa 1	Etapa 2	Etapa 3	Etapa 4
IR ← (PC)	PC ← PC + 2 Dst ← (Rs)	AluStart ← 1 (Rs) ← Result	Rs ← Rs + 2

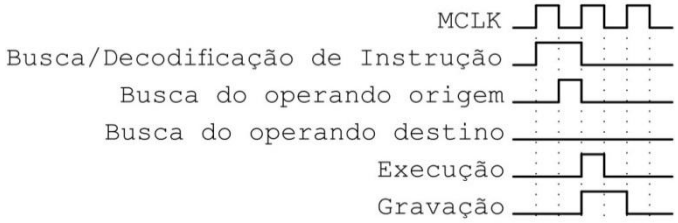

Figura 26 - Instruções de um operando com modo de endereçamento As=11b.

- **Instruções de dois operandos com modo de endereçamento As=00b e Ad=0**

Etapa 1	Etapa 2	Etapa 3	Etapa 4
IR ← (PC)	PC ← PC + 2 Src ← Rs	Dst ← Rd	AluStart ← 1 Rd ← Result

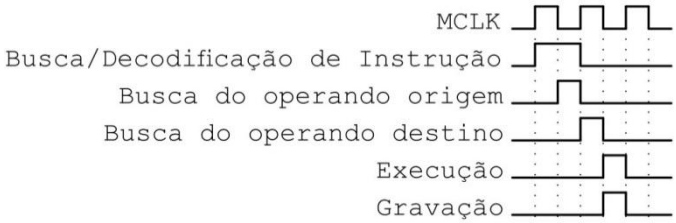

Figura 27 - Instruções de dois operandos com modo de endereçamento As=00b e Ad=0.

- **Instruções de dois operandos com modo de endereçamento As=00b e Ad=1**

Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5
IR ← (PC)	PC ← PC + 2 Src ← Rs	Ofs ← (PC)	PC ← PC + 2 Dst ← (Rd+Ofs)	AluStart ← 1 (Rd+Ofs) ← Result

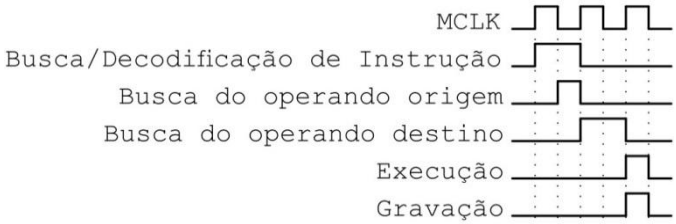

Figura 28 - Instruções de dois operandos com modo de endereçamento As=00b e Ad=1.

- **Instruções de dois operandos com modo de endereçamento As=01b e Ad=0**

Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5
IR ← (PC)	PC ← PC + 2 Ofs ← (PC)	PC ← PC + 2 Src ← (Ofs+Rs)	Dst ← Rd	AluStart← 1 Rd ← Result

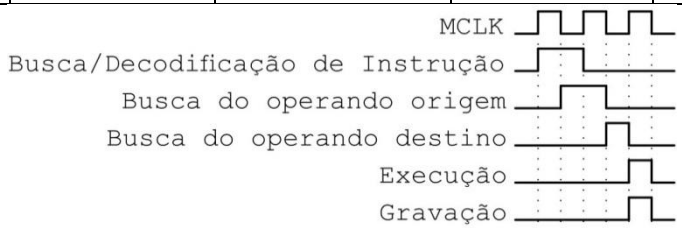

Figura 29 - Instruções de dois operandos com modo de endereçamento As=01b e Ad=0.

- **Instruções de dois operandos com modo de endereçamento As=01b e Ad=1**

Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5
IR←(PC)	PC←PC+2 Ofs←(PC)	Src←(Rs+Ofs)	Ofs←(PC+2)	PC←PC+2 Dst←(Rd+Ofs)

Etapa 6	Etapa 7
AluStart←1 (Rd+Ofs) ←Result	PC←PC+2

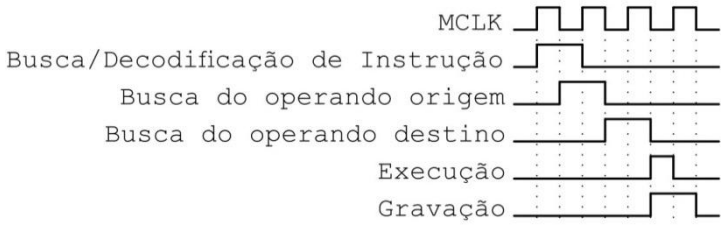

Figura 30 - Instruções de dois operandos com modo de endereçamento As=01b e Ad=1.

- **Instruções de dois operandos com modo de endereçamento As=10b e Ad=0**

Etapa 1	Etapa 2	Etapa 3	Etapa 4
IR ← (PC)	PC ← PC + 2 Src ← (Rs)	Dst ← Rd	AluStart ← 1 Rd ← Result

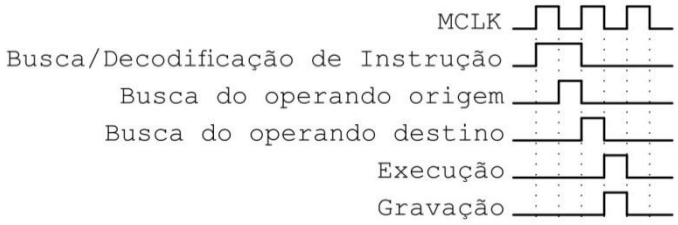

Figura 31 - Instruções de dois operandos com modo de endereçamento As=10b e Ad=0.

- **Instruções de dois operandos com modo de endereçamento As=10b e Ad=1**

Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5
IR ← (PC)	PC ← PC + 2 Ofs ← (PC)	Src ← (Rs)	PC ← PC + 2 Dst ← (Rd+Ofs)	AluStart← 1 (Rd+Ofs)← Result

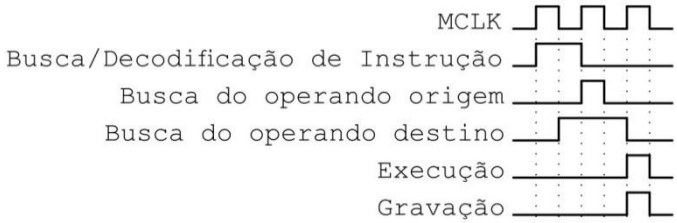

Figura 32 - Instruções de dois operandos com modo de endereçamento As=10b e Ad=1.

- **Instruções de dois operandos com modo de endereçamento As=11b e Ad=0**

Etapa 1	Etapa 2	Etapa 3	Etapa 4
IR ← (PC)	PC ← PC + 2 Src ← (Rs)	Dst ← Rd Rs ← Rs+2	AluStart ← 1 Rd ← Result

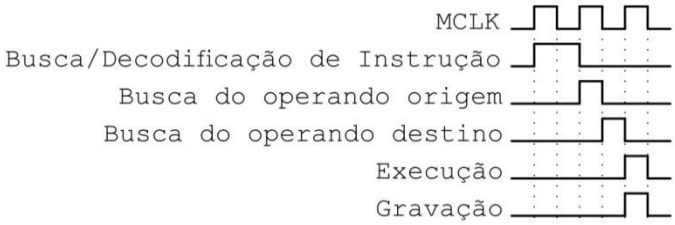

Figura 33 - Instruções de dois operandos com modo de endereçamento As=11b e Ad=0.

- **Instruções de dois operandos com modo de endereçamento As=11b e Ad=1**

Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5
IR ← (PC)	PC ← PC + 2 Ofs ← (PC)	PC ← PC + 2 Src ← (Rs)	Ofs ← (Rd+Ofs) Rs ← Rs + 2	AluStart← 1 (Rd+Ofs)← Result

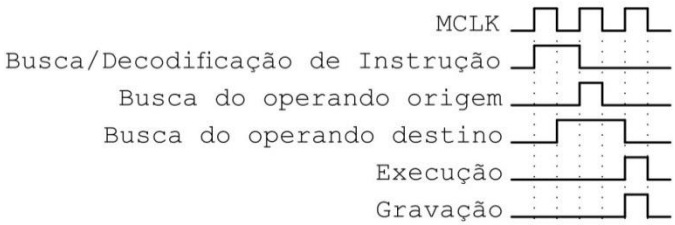

Figura 34 - Instruções de dois operandos com modo de endereçamento As=11b e Ad=1.

3.2. Port

O circuito do *port* implementado desempenha as mesmas funções exercidas pelos *ports* do MSP430. Foram implementados *ports* bidirecionais de 8 bits controlados individualmente, todos eles com suporte à interrupção e direção controlada através um *buffer tri-state*.

A figura 35 mostra o diagrama de sinais do *port* implementado.

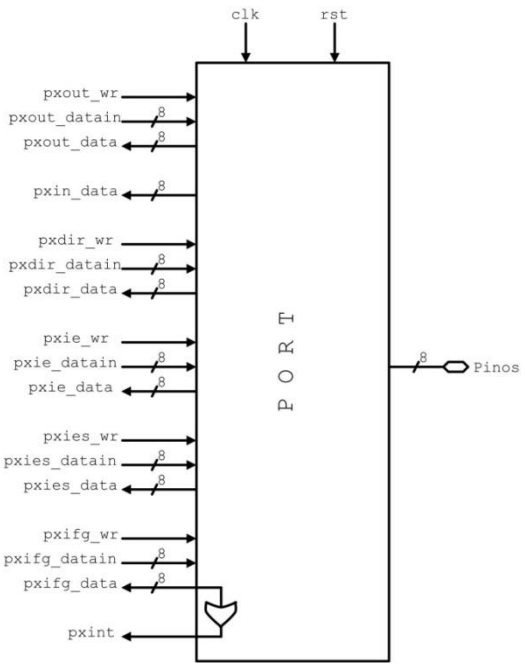

Figura 35 - Diagrama de sinais do port implementado.

Os sinais à esquerda da figura 35 são os sinais que devem ser conectados à CPU através do barramento. Consistem em sinais de leitura e escrita dos registradores, exceto o sinal "pxint". Este sinal se resume a uma lógica "ou" entre todos os bits do registrador "PxIFG", sendo assim setado sempre que houver alguma interrupção pendente no port. Esse sinal deve ser conectado diretamente à entrada de interrupções da unidade de controle da CPU.

A figura 36 mostra a interface de entrada e saída de dados dos pinos do *port* projetado. Ela é constituída basicamente por um *buffer tri-state* controlado pelo registrador PxDIR e dois *flip-flops*, denominados A e B, ambos sensíveis a borda de subida do sinal de *clock*. O *flip-flop* A é responsável por armazenar temporariamente o sinal de entrada do pino, enquanto o *flip-flop* B armazena temporariamente o sinal de saída.

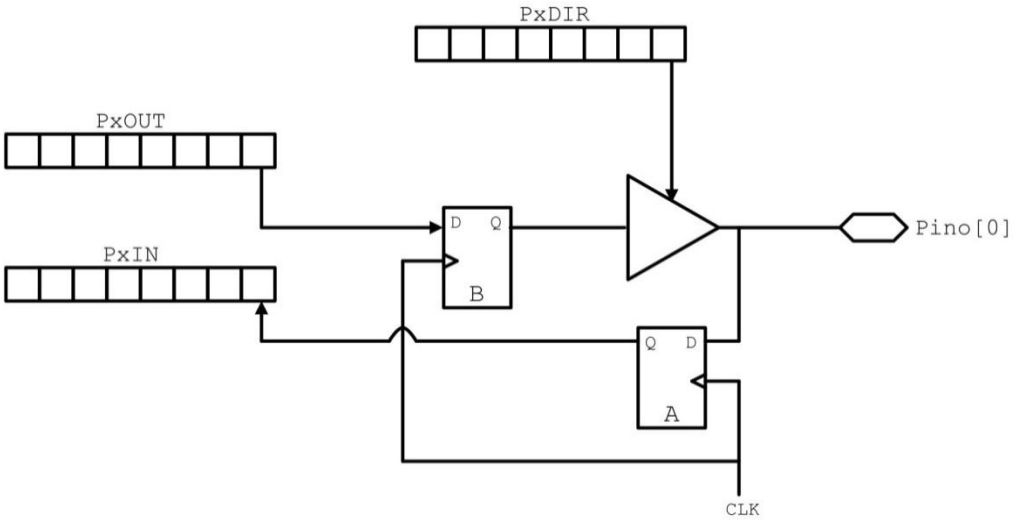

Figura 36 - Interface de entrada e saída do port.

Quando um determinado bit do registrador PxDIR é setado, o *buffer* associado a ele passa a conduzir. Então em cada borda de subida do sinal de *clock*, a saída do *flip-flop* B assume o valor de seu bit associado no registrador PxOUT e esse valor é levado para o pino através do *buffer*.

De forma semelhante, quando um determinado bit do registrador PxDIR é resetado, o *buffer tri-state* associado a ele assume o estado de alta impedância. Assim, a cada borda de subida do sinal de *clock*, o valor presente no pino é transferido através do *flip-flop A* para o registrador PxIN.

Como o registrador PxIN é gravado na borda de subida do *clock*, as interrupções geradas pelo *port* também são. Elas são geradas quando ocorre transição positiva ou negativa de um sinal de entrada, dependendo do bit correspondente no registrador PxIES. Para auxiliar na detecção de bordas, foi usado um registrador interno ao *port* denominado "px_prev". Esse registrador tem a função de armazenar o valor de PxIN do ciclo anterior de *clock*. Para isso, ele copia o valor de PxIN na borda de descida do sinal de *clock*, como mostra o diagrama da figura 37.

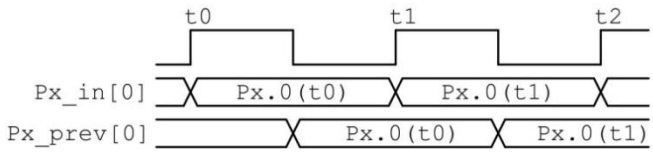

Figura 37 - Diagrama do registrador interno "Px_prev".

Em cada transição positiva do sinal de *clock*, o conteúdo do registrador PxIN é comparado com o conteúdo do registrador "px_prev" para verificar se houve variação no sinal de entrada. Além da variação do sinal de entrada, para que haja solicitação de interrupção, é necessário que o registrador PxIE habilite a interrupção no pino correspondente e o tipo de transição seja o mesmo indicado pelo registrador PxIES. A tabela 11 mostra as condições necessárias para que haja solicitação de interrupção, por meio de um pino "n".

Tabela 11 - Condições de interrupção do port

Tipo	PxIE[n]	PxIES[n]	Px_prev[n]	PxIN[n]
Borda de Subida	1	0	0	1
Borda de Descida	1	1	1	0

Como pode ser observado na tabela 11, as condições para que haja solicitação de interrupção podem ser resumidas em:

- A interrupção daquele pino estar habilitada no registrador PxIE;
- Ter havido transição de estado naquele pino e
- A transição que houve é aquela definida no registrador PxIES.

Caso essas condições tenham sido satisfeitas, o respectivo bit do registrador PxIFG assume valor lógico alto indicando que há interrupção pendente.

3.3. Módulo básico de clock

O circuito do módulo básico de *clock* implementado neste projeto é uma versão bastante simplificada de seu circuito equivalente nos microcontroladores MSP430. Essa simplificação se deve ao fato de que o módulo básico de *clock* funciona em torno de seu oscilador analógico, o DCO.

A natureza analógica deste circuito impossibilita que ele seja implementado em uma FPGA. Para contornar essa dificuldade, optou-se por obter os sinais de *clock* a partir de fontes externas à FPGA. Para isso, o circuito implementado conta com duas entradas de *clock*, uma de alta e outra de baixa frequência.

O circuito implementado é mostrado na figura 38.

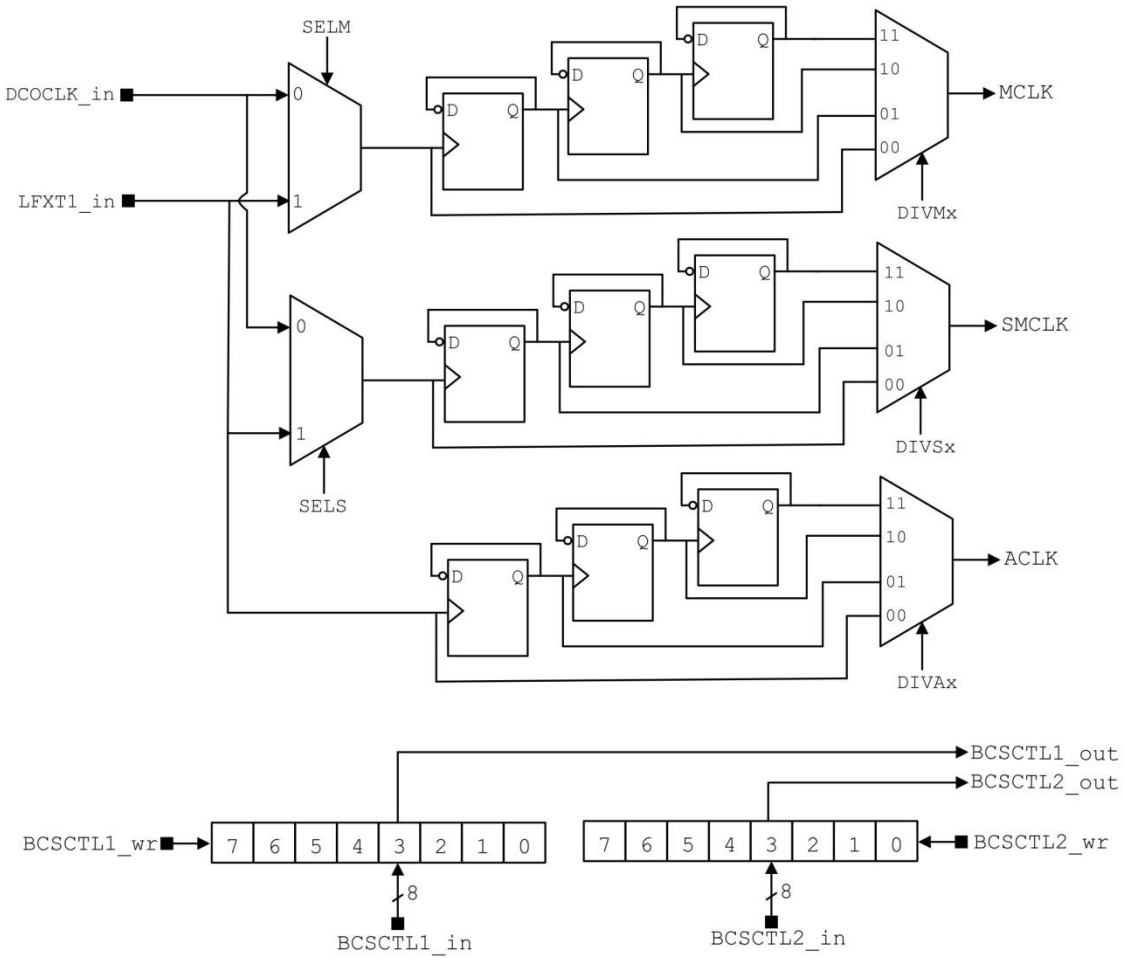

Figura 38 - Circuito implementado do módulo básico de clock.

Como pode ser observado na figura 38, o circuito implementado conta com duas entradas de *clock*. A entrada "DCOCLK_in" é usada para o sinal de *clock* de alta frequência. Sua função é suprir a ausência do oscilador interno DCO. A entrada "LFXT1_in" é usada para o sinal de baixa frequência.

Sem o oscilador analógico, o circuito implementado se resume a duas funções, selecionar a origem dos sinais e dividir sua frequência.

Os multiplexadores de entrada têm a função de selecionar a origem dos sinais MCLK e SMCLK através dos bits SELM e SELS do registrador BCSCTL2. Como pode ser visto na figura 38, o sinal ACLK não possui seleção de origem, sua fonte é sempre o sinal de baixa frequência.

Os *flip-flops* presentes neste circuito estão dispostos na configuração de divisor de frequência. Sua função é dividir a frequência do sinal selecionado por dois, quatro e oito. Esses três sinais com frequência dividida juntamente com o sinal original selecionado pelos

multiplexadores de entrada são ligados aos multiplexadores de saída, cuja função é selecionar, através dos bits de controle DIVMx, DIVSx e DIVAx quais destes sinais serão os *clocks* MCLK, SMCLK e ACLK do sistema.

3.4. Watchdog timer

A figura 39 mostra o diagrama de blocos do circuito do implementado.

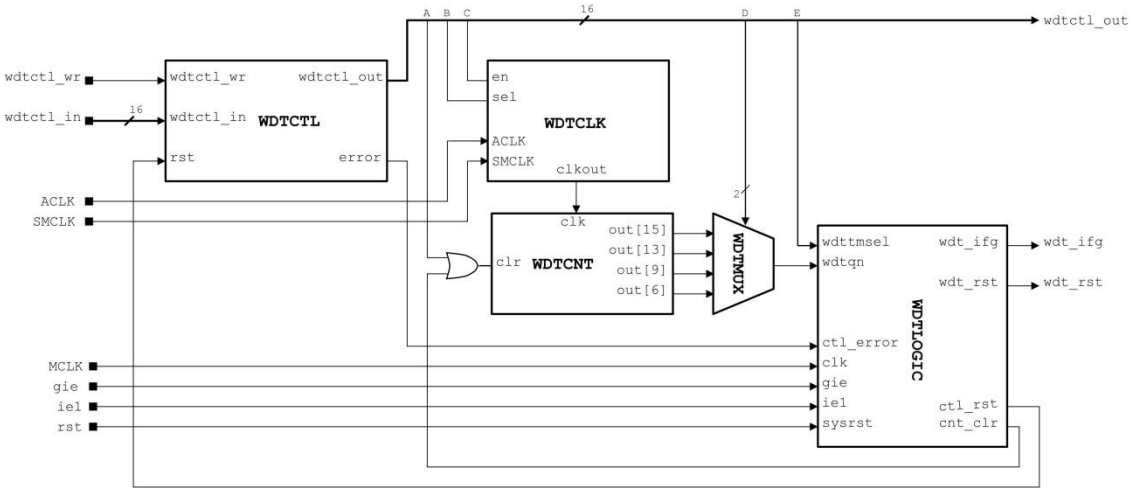

Figura 39 - Circuito implementado do watchdog timer.

Como pode ser visto pela figura 39, o projeto do *watchdog timer* foi dividido em quatro módulos básicos, denominados:

- WDTCTL;
- WDTCLK;
- WDTCNT e
- WDTLOGIC.

O primeiro módulo contém o registrador WDTCTL que é responsável pelo controle do periférico e sua função é gerencia-lo. Possui três sinais de entrada, uma entrada de dados de 16 bits, denominada “wdtctl_in”, um sinal de controle que habilita a escrita no registrador quando setado, “wdtctl_wr”, e um sinal de reset, “rst”, usado para colocar o registrador em seu estado inicial. Este módulo possui duas saídas, uma saída de 16 bits contendo o conteúdo do registrador WDTCTL, chamada “wdtctl_out” e um sinal de controle, chamado de “error”, que é setado sempre há violação da senha durante a escrita no registrador.

O módulo WDTCNT é o contador de 16 bits, o componente mais importante do *watchdog timer*. O incremento do contador é realizado sempre na transição positiva do sinal "clk". O sinal "clr" reinicia a contagem sempre que está em nível lógico alto. A contagem pode ser reiniciada quando atinge seu valor máximo ou quando o bit WDTCNTCL (indicado na figura 39 pela derivação "A" do sinal "wdtctl_out") é setado. Os bits 15, 13, 9 e 6 do contador, associados respectivamente às contagens 32.768, 8.192, 512 e 64, são ligados ao multiplexador WDTMUX.

O multiplexador WDTMUX tem a função de selecionar quantos pulsos devem ser contados para que haja solicitação de interrupção ou *reset*. Seu controle é feito por meio dos bits WDTISx do registrador WDTCTL.

O módulo WDTCLK tem a função de selecionar a origem do sinal de *clock* que excita o contador WDTCNT. A seleção é feita através da entrada "sel" que é conectada ao bit de controle WDTSSEL.

O módulo WDTLOGIC contém a lógica de controle responsável pela geração dos sinais de interrupção e *reset*, bem como os sinais que reiniciam a contagem de WDTCNT e resetam o valor do registrador de controle WDTCTL. A lógica presente neste módulo é representada pelo fluxograma da figura 40.

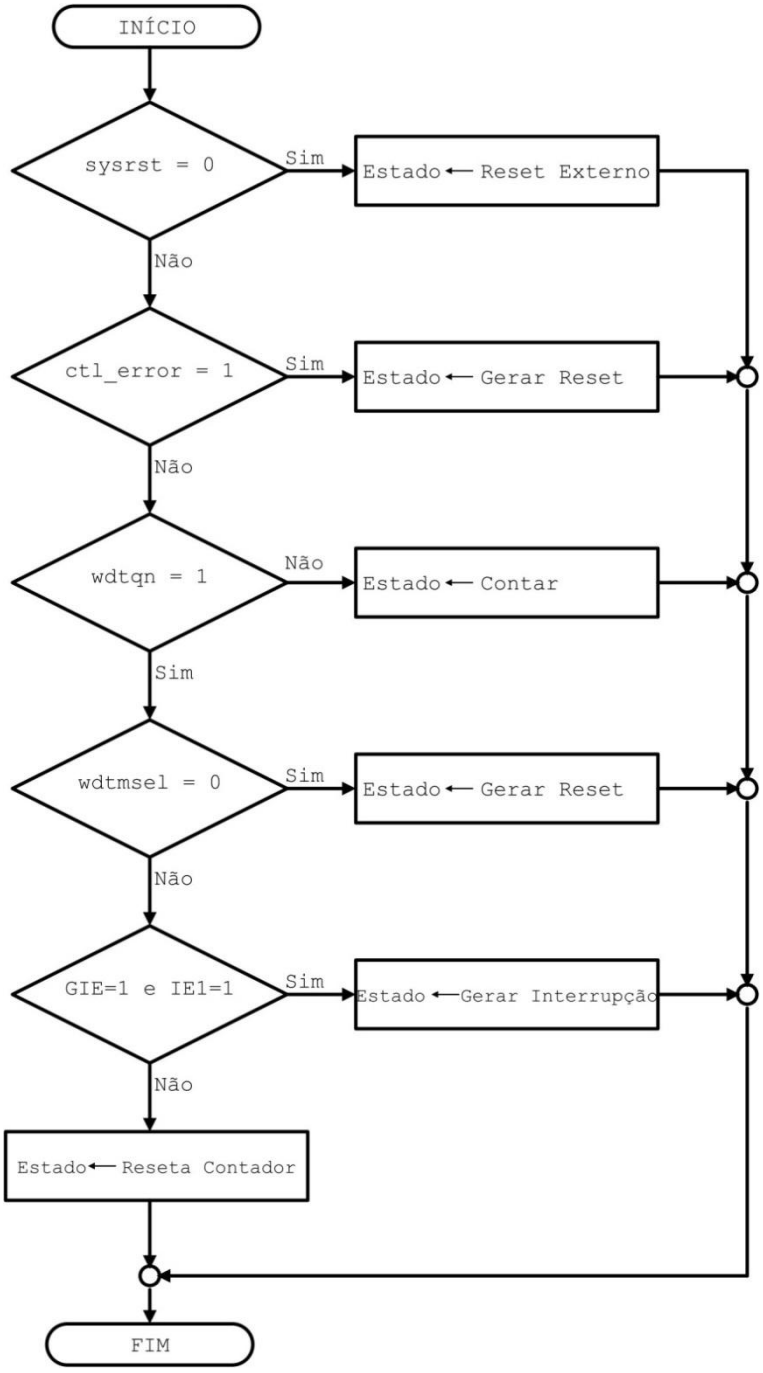

Figura 40 - Lógica do módulo WDTLOGIC do watchdog timer.

Os “estados” mencionados no fluxograma da figura 40 referem-se a um conjunto específico de saída descritos pela tabela 12.

Tabela 12 - Estados do watchdog timer

		Sinais de Saída			
		wdt_rst	**wdt_ifg**	**cnt_clr**	**ctl_rst**
Estados	**Reset externo**	1	0	1	0
	Gerar reset	0	0	1	0
	Contar	1	0	0	1
	Gerar interrupção	1	1	1	1
	Reseta contador	1	0	1	1

O estado “reset externo” é acionado sempre que o microcontrolador é reiniciado. Nesse estado, o contador é reiniciado e o registrador de controle assume seu valor padrão.

O estado “gerar reset” acontece quando o dispositivo está configurado para funcionar no modo *watchdog* e a contagem atinge seu valor máximo ou quando há violação na escrita do registrador de controle. Nesse estado, o contador é reiniciado, o registrador de controle assume seu valor padrão e o sinal de *reset* é levado ao nível lógico baixo, solicitando reinicialização do sistema.

O estado “contar” apenas incrementa o valor do contador a cada pulso de *clock*.

O estado “gerar interrupção” ocorre quando o dispositivo está configurado para operar no modo temporizador e a contagem atinge seu máximo. Nesse estado, o contador é reiniciado e o sinal de interrupção é setado, solicitando interrupção.

Por fim, o estado “Reseta contador” ocorre quando o dispositivo está configurado como temporizador e o contador alcança o valor máximo, mas as interrupções estão desabilitadas. Nesse, estado o contador é resetado, mas nenhuma interrupção ou reset é solicitado.

3.5. Timer A

O circuito implementado do temporizador “Timer A” é muito semelhante ao circuito mostrado na figura 10. O circuito implementado no projeto, também foi dividido em dois blocos: o bloco do registrador TAR e o bloco de captura/comparação, permitindo que a um bloco TAR sejam adicionados tantos blocos de captura/comparação quanto sejam necessários. O bloco TAR implementado é mostrado na figura 41.

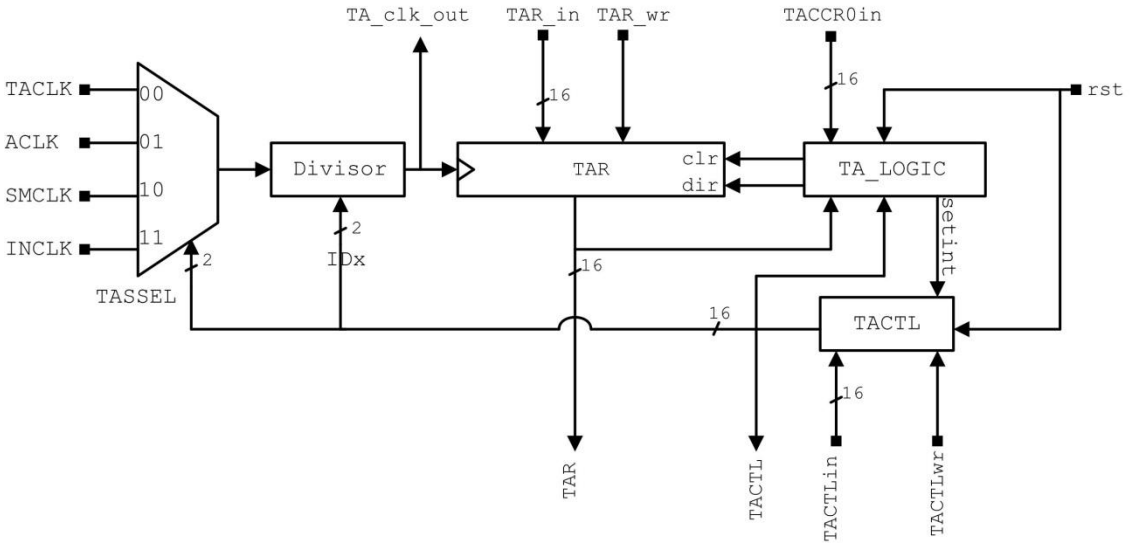

Figura 41 - Bloco TAR.

O controle desse bloco é feito através do circuito chamado “TA_LOGIC”, ele é responsável por definir a direção de contagem do registrador TAR, reinicia-lo e ativar o *flag* de interrupção, do registrador TACTL. O multiplexador é responsável por definir a origem do sinal que excitará o contador TAR e o divisor divide a frequência do sinal de entrada por dois, quatro ou oito.

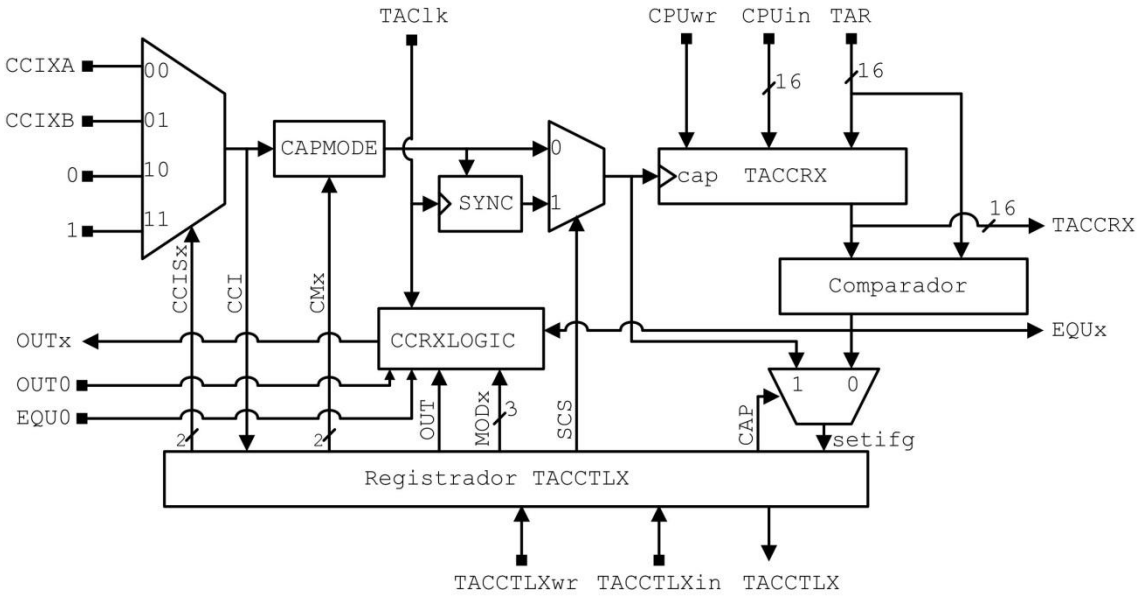

Figura 42 - Bloco de captura/comparação do Timer A.

O bloco de captura/comparação é mostrado na figura 42. O multiplexador de entrada é responsável por selecionar o sinal de captura quando o bloco estiver configurado para este fim. O circuito CAPMODE monitora o sinal de entrada e quando o tipo de transição selecionada pelos bits CM for satisfeita, ele gera um pulso positivo, ordenando que a captura seja feita. O circuito SYNC faz a sincronia desse pulso positivo com o *clock* do bloco TAR. Então, o sinal sincronizado e o não-sincronizado podem ser selecionados através de um multiplexador controlado pelo *flag* SCS cuja saída é ligada ao sinal de captura do registrador TACCRx.

Quando o bloco estiver configurado para operar em modo de comparação, o valor a ser comparado deve ser armazenado no registrador TACCRx. Quando este valor for alcançado pelo registrador TAR, a saída do comparador será levada ao nível lógico alto indicando que o registrador TAR alcançou o valor a ser comparado.

O multiplexador controlado pelo sinal CAP é responsável por gerar o sinal que solicita a interrupção do bloco. Quando CAP=0 (modo comparação), o sinal responsável por setar o flag de interrupção é a saída do comparador. Quando CAP=1, o sinal responsável por setar o flag de interrupção é o mesmo que excita a captura no registrador TACCRx.

Por fim, o bloco denominado CCRXLOGIC na figura 42, é simplesmente uma lógica responsável por gerar o sinal de saída OUTx de acordo com a opção escolhida através dos bits MODx do registrador de controle.

O bloco de captura/comparação pode ser configurado para funcionar de duas maneiras diferentes. O modo de comparação é ilustrado pela figura 43.

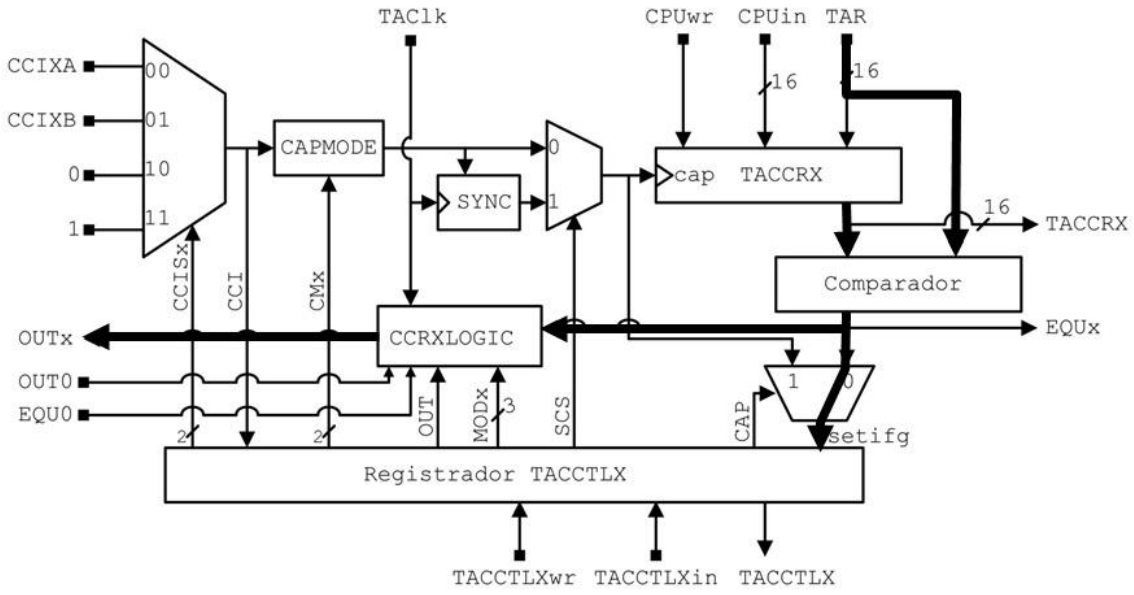

Figura 43 - Timer A em modo de comparação.

Neste modo, o valor a ser comparado deve ser carregado pelo usuário no registrador TACCRx. O valor armazenado por esse registrador, juntamente com o valor de contagem do registrador TAR são levados à entrada do comparador. Quando o valor armazenado por esses dois registradores tornar-se igual, a saída do comparador passará do nível lógico baixo para o nível alto, solicitando interrupção e gerando o sinal OUTx configurado pelo usuário.

O modo de captura é ilustrado pela figura 44. Neste modo, o sinal de origem e o tipo de transição que disparam a captura devem ser selecionados pelo usuário via software. Quando a transição selecionada ocorre na entrada do detector de bordas CAPMODE, um pulso positivo é gerado em sua saída. Esse pulso é levado então ao registrador TACCRX, que neste momento armazena o valor do registrador TAR, caracterizando a captura. Este mesmo pulso então é usado como sinal de solicitação de interrupção e serve para setar o bit de interrupção no registrador TACCTLX.

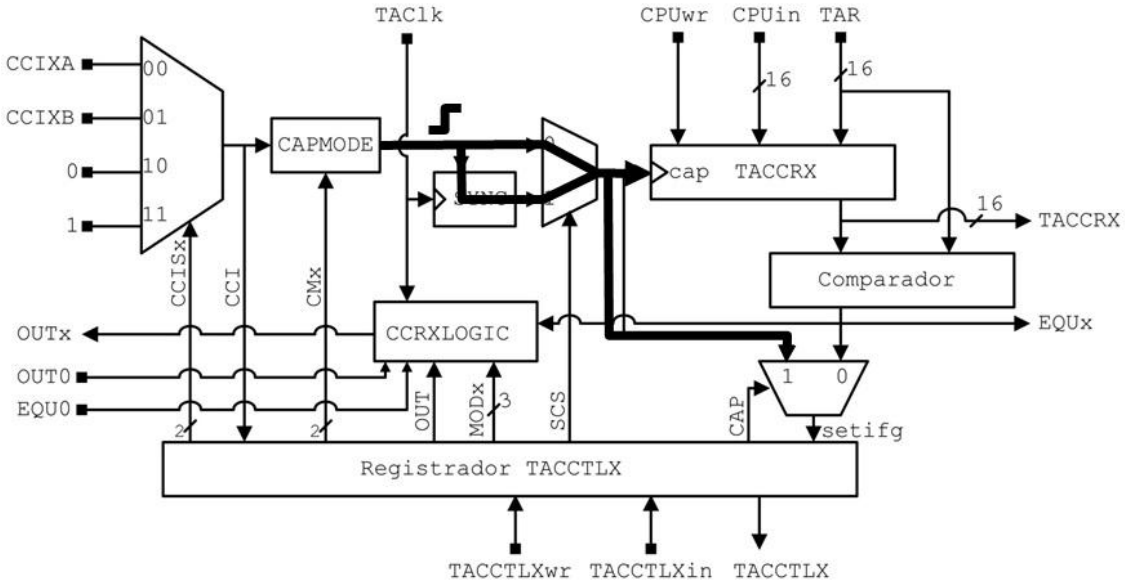

Figura 44 - Timer A em modo de captura.

4. RESULTADOS

Este capítulo é dedicado a mostrar os resultados obtidos por meio de simulações feitas em todos os módulos implementados neste projeto. Todas as simulações foram feitas com auxílio do software ModelSim, desenvolvido pela Mentor Graphics.

4.1. ULA

As simulações das operações da ULA foram feitas forçando-se os valores de operandos e operação na entrada e lendo-se o resultado fornecido pela saída "Result" e demais flags. Os resultados foram então comparados com os obtidos fazendo-se a mesma operação no microcontrolador MSP430G2231.

Diante da impossibilidade de se cobrir todos os resultados possíveis, será mostrado um resultado de simulação para cada operação.

- **Teste da operação: ADD**

  Operação realizada: 1234h + 5678h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 68ACh

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

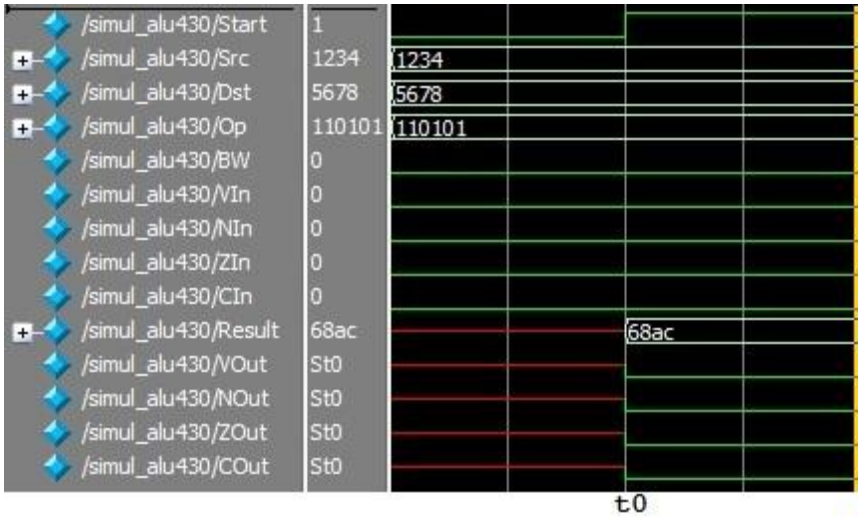

Figura 45 - Resultado da simulação da operação ADD.

Pode-se observar na figura 45 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação ADD.

- **Teste da operação: ADDC**

  Operação realizada: 1234h + 5678h com carry de entrada igual a 1.

  Flags de entrada: C = 1, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 68ADh

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

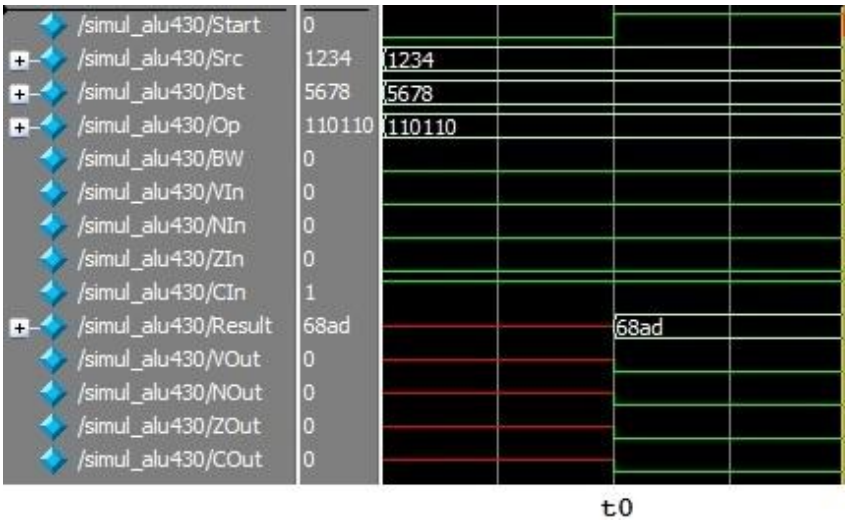

Figura 46 - Resultado da simulação da operação ADDC.

Pode-se observar na figura 46 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação ADDC.

- **Teste da operação: SUB**

  Operação realizada: 1234h - 1111h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0123h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

Figura 47 - Resultado da simulação da operação SUB.

Pode-se observar na figura 47 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação SUB.

- **Teste da operação: SUBC**

  Operação realizada: 1234h - 1111h com carry de entrada igual a 0

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0123h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

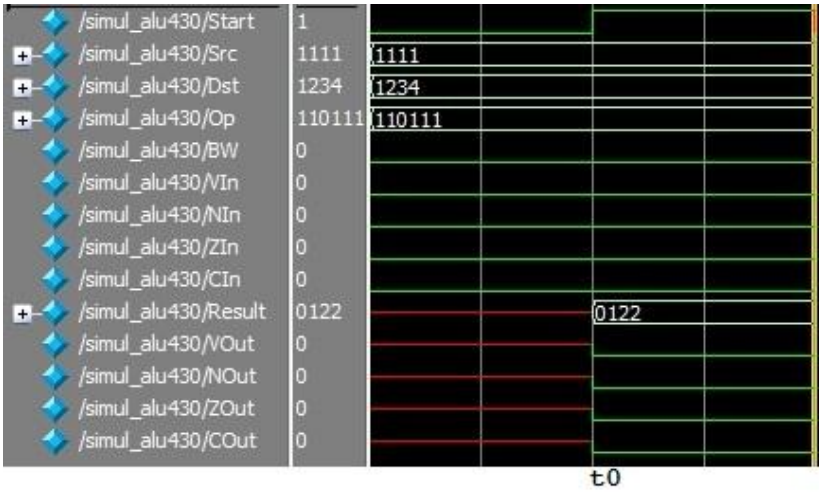

Figura 48 - Resultado da simulação da operação SUBC.

Pode-se observar na figura 48 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação SUBC.

- **Teste da operação: CMP**

  Operação realizada: 0123h - 1234h com instrução CMP

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0123h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 1, V = 0.

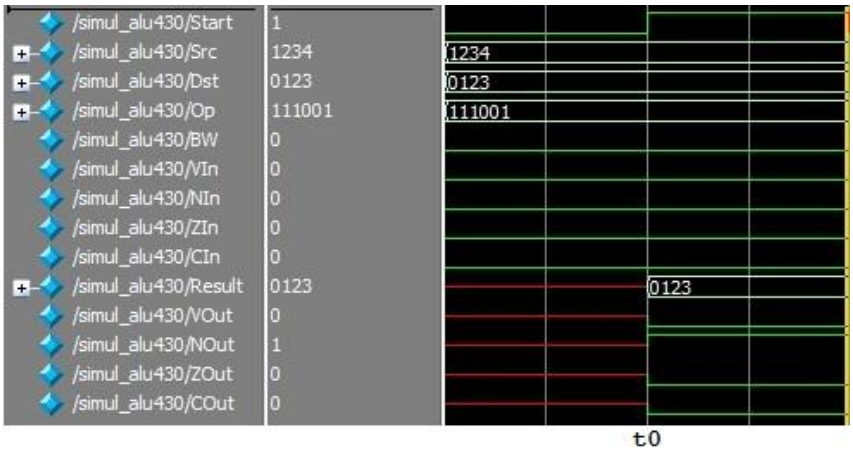

Figura 49 - Resultado da simulação da operação CMP.

Pode-se observar na figura 49 que no instante t_0, quando ocorre a transição positiva do sinal "Start", os *flags* de saída são atualizados, e a saída "Result" assume o valor do registrador DST caracterizando a operação CMP.

- **Teste da operação: DADD**

  Operação realizada: 0123 + 1234

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 1357

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

Figura 50 - Resultado da simulação da operação DADD.

Pode-se observar na figura 50 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação DADD.

- **Teste da operação: AND**

  Operação realizada: 0F0Fh AND 1234h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0204h

  Flags de saída observados no MSP430: C = 1, Z = 0, N = 0, V = 0.

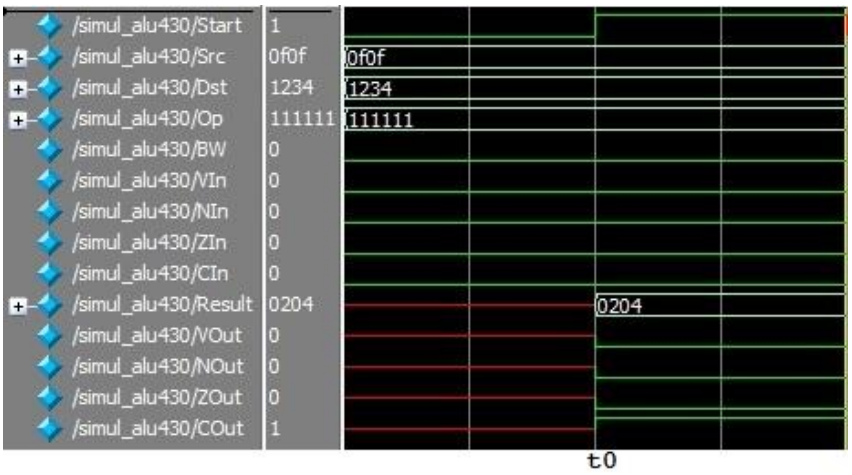

Figura 51 - Resultado da simulação da operação AND.

Pode-se observar na figura 51 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação AND.

- **Teste da operação: BIT**

  Operação realizada: 0F0Fh BIT 1234h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 1234h

  Flags de saída observados no MSP430: C = 1, Z = 0, N = 0, V = 0.

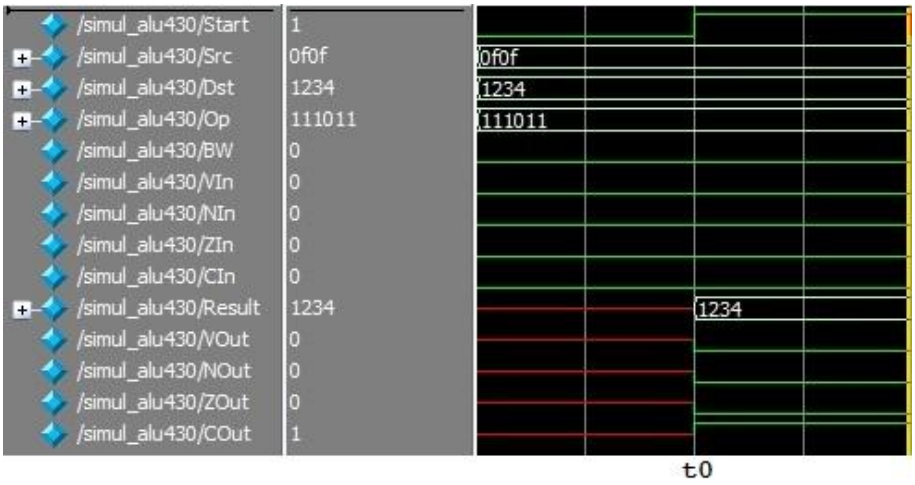

Figura 52 - Resultado da simulação da operação BIT.

Pode-se observar na figura 52 que no instante t_0, quando ocorre a transição positiva do sinal “Start”, os *flags* de saída são atualizados, e a saída “Result” assume o valor do registrador DST caracterizando a operação BIT.

- **Teste da operação: BIC**

  Operação realizada: 0F0Fh BIC 1234h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 1030h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

Figura 53 - Resultado da simulação da operação BIC

Pode-se observar na figura 53 que no instante t_0, quando ocorre a transição positiva do sinal “Start”, o sinal “Result” e os *flags* de saída são atualizados, caracterizando a operação BIC.

- **Teste da operação: BIS**

  Operação realizada: 0F0Fh BIS 1234h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 1F3Fh

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

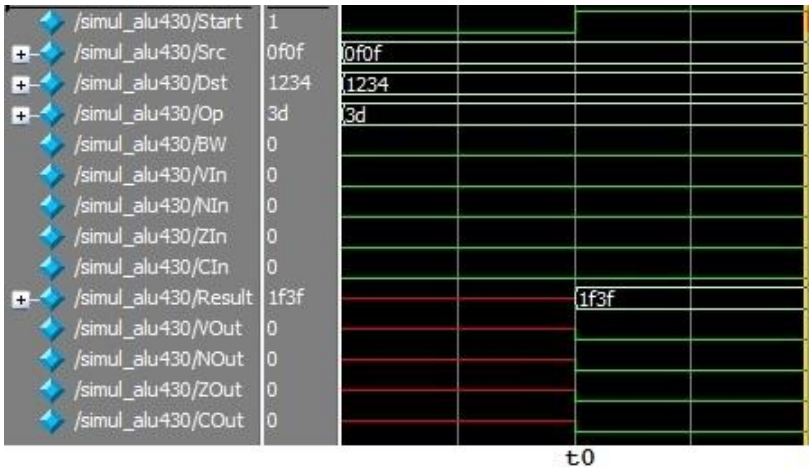

Figura 54 - Resultado da simulação da operação BIS.

Pode-se observar na figura 54 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação BIS.

- **Teste da operação: XOR**

  Operação realizada: 0F0Fh XOR 0FFFFh

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: F0F0h

  Flags de saída observados no MSP430: C = 1, Z = 0, N = 1, V = 0.

Figura 55 - Resultado da simulação da operação XOR.

Pode-se observar na figura 55 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação XOR.

- **Teste da operação: RRC**

  Operação realizada: RRC 0F0Fh

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0787h

  Flags de saída observados no MSP430: C = 1, Z = 0, N = 0, V = 0.

Figura 56 - Resultado da simulação da operação RRC.

Pode-se observar na figura 56 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação RRC.

- **Teste da operação: RRA**

  Operação realizada: RRA FF00h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: FF80h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 1, V = 0.

Figura 57 - Resultado da simulação da operação RRA.

Pode-se observar na figura 57 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação RRA.

- **Teste da operação: SWPB**

  Operação realizada: SWPB 1234h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 3412h

  Flags de saída observados no MSP430: C = 0, Z = 0, N = 0, V = 0.

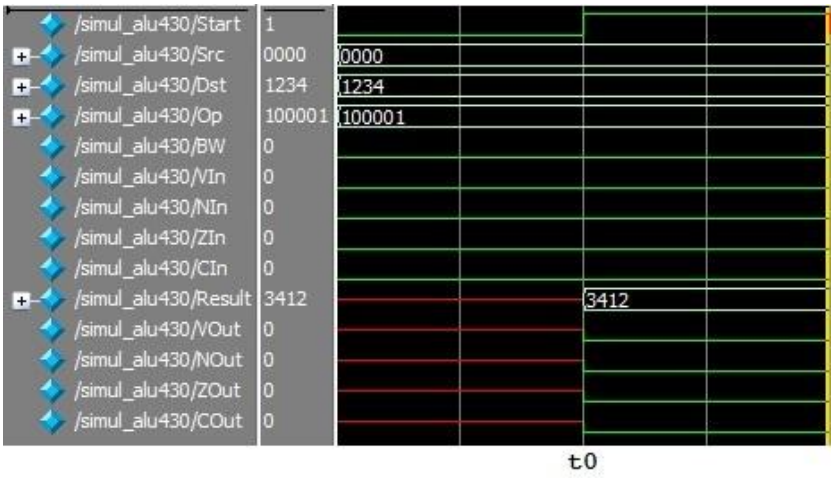

Figura 58 - Resultado da simulação da operação SWPB.

Pode-se observar na figura 58 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" é atualizado, caracterizando a operação SWPB.

- **Teste da operação: SXT**

  Operação realizada: SXT FF12h

  Flags de entrada: C = 0, Z = 0, N = 0, V = 0.

  Resultado observado no MSP430: 0012h

  Flags de saída observados no MSP430: C = 1, Z = 0, N = 0, V = 0.

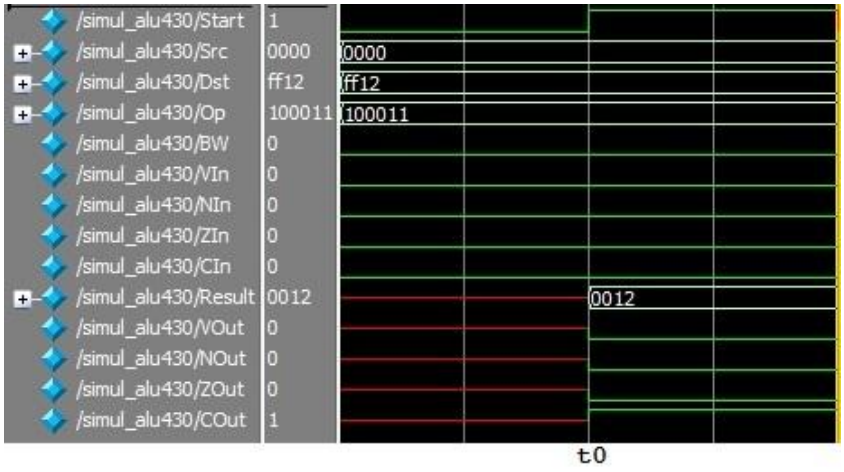

Figura 59 - Resultado da simulação da operação SXT.

Pode-se observar na figura 59 que no instante t_0, quando ocorre a transição positiva do sinal "Start", o sinal "Result" e os *flags* de saída são atualizados, caracterizando a operação SXT.

4.2. CPU

- **Instruções de um operando com modo de endereçamento As=00b**

Para simular este tipo de instrução, foi feita a operação `SWPB R4`, com R4 carregado previamente com o valor 0ABCDh.

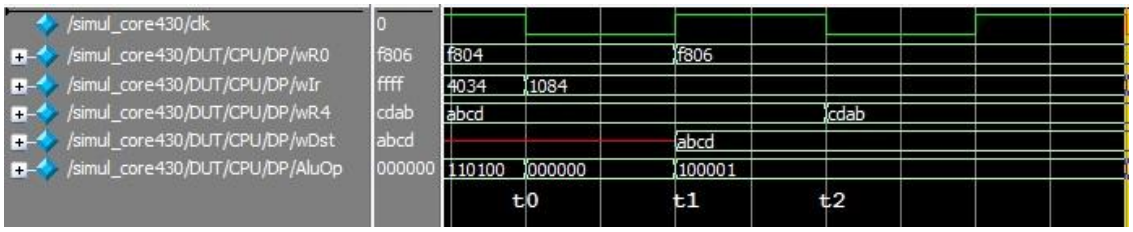

Figura 60 - Resultado da simulação de instruções de um operando com As=00.

Nota-se, na figura 60, que no instante indicado por t_0 ocorre a busca da instrução que é carregada no registrador IR. Em t_1 o corre o ciclo de busca do operando de origem que é carregado no registrador DST. Neste instante o registrador R0/PC é também incrementado. Em t_2 a operação é executada e gravada no registrador destino R4.

- **Instruções de um operando com modo de endereçamento As=01b**

Para simular este tipo de instrução, foi feita a operação `SWPB 4(R4)`, com R4 carregado inicialmente com o endereço 0200h e o endereço de memória 0204h carregado com o 0ABCDh.

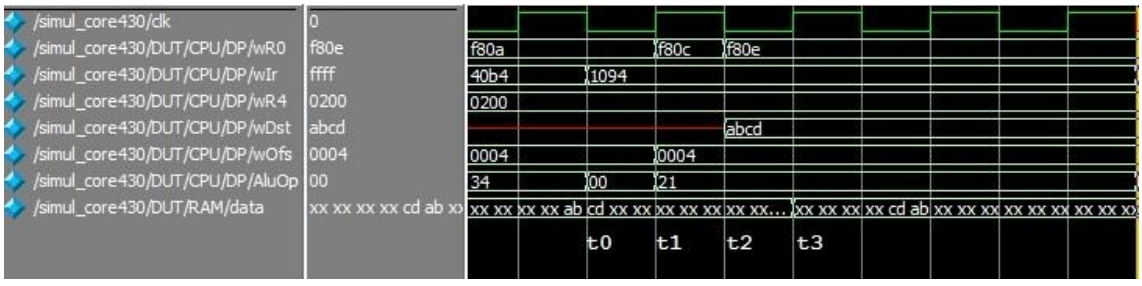

Figura 61 - Resultado da simulação de instruções de um operando com As=01.

Em t_0, inicia-se o ciclo de busca e a instrução 1094h é carregada no registrador de instruções IR. Em t_1, o registrador PC é incrementado e seu conteúdo (0004h) é carregado no registrador "OFS/TEMP". Em t_2, o ciclo de busca de operando é encerrado quando o operando 0ABCDh é carregado do endereço 0204h no registrador DST. Finalmente em t_3, a operação é executada e o resultado é gravado no endereço 0204h da memória de dados.

- **Instruções de um operando com modo de endereçamento As=10b**

Para simular este tipo de instrução, foi feita a operação `SWPB @R4`, com R4 carregado inicialmente com o endereço 0200h e o endereço de memória 0200h carregado com o 0ABCDh.

Figura 62 - Resultado da simulação de instruções de um operando com As=10.

No instante t_0, a instrução 10A4h é carregada no registrador de instruções. Em t_1, o registrador R0/PC é incrementado e em t_2, a operação é executada e armazenada em seu destino.

- **Instruções de um operando com modo de endereçamento As=11b**

Para simular este tipo de instrução, foi feita a operação `SWPB @R4+`, com R4 carregado inicialmente com o endereço 0200h e o endereço de memória 0200h carregado com o ABCDh.

Figura 63 - Resultado da simulação de instruções de um operando com As=11.

Em t_0, a instrução é carregada no registrador IR. Em t_1, o registrador PC é incrementado e o operando é carregado no registrador DST. Em t2 a operação é realizada e o resultado é armazenado no endereço 0200h da memória. Em t_3, o registrador R4 é incrementado.

- **Instruções de dois operandos com modo de endereçamento As=00b e Ad=0b**

Para simular este tipo de instrução, foi feita a operação `ADD.W R4,R5`, com o registrador R4 carregado previamente com o valor 0200h e o registrado R5 carregado com o valor 0100h.

Figura 64 - Resultado da simulação de instruções de um operando com As=00 e Ad=0.

Em t_0, a instrução é carregada no registrador de instruções. Em t_1, o valor do registrador R4 é carregado no registrador SRC. Em t_2, o valor do registrador R5 é carregado no registrador DST. Em t_3, a operação é realizada e o resultado é gravado no registrador R5.

- **Instruções de dois operandos com modo de endereçamento As=00b e Ad=1b**

Para simular este tipo de instrução, foi feita a operação `ADD.W R5,2(R4)`, com o registrador R4 carregado previamente com o endereço 0200h, o registrado R5 carregado com o valor 0100h e a posição 0202h da memória de dados com o valor 0300h.

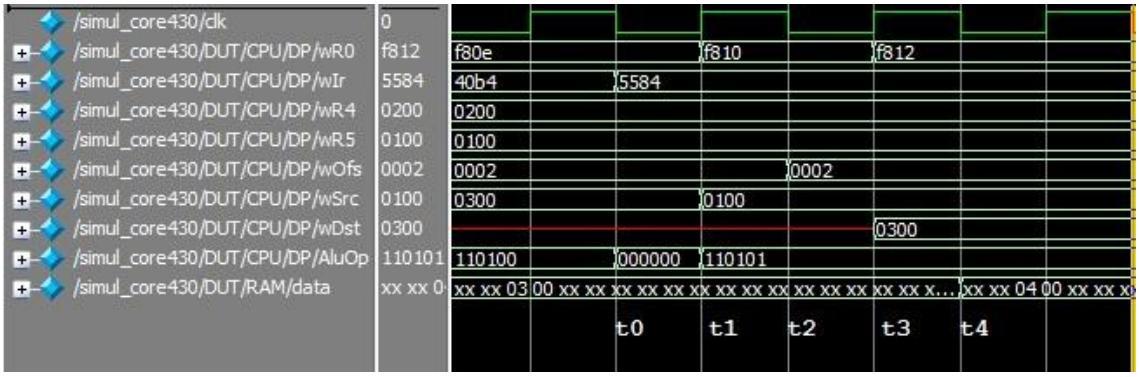

Figura 65 - Resultado da simulação de instruções de um operando com As=00 e Ad=1.

No instante t_0, a instrução é carregada no registrador de instruções. Em t_1, o conteúdo de R5 é carregado no registrador SRC e o registrador PC é incrementado. Em t_2 o *offset* 0002h, apontado por PC, é carregado no registrador OFS/TEMP. Em t_3, o valor 0300h que está contido

no endereço apontado por R4+2 é carregado no registrador DST. Finalmente, em t_4 a operação é realizada e o resultado é gravado na memória de dados, no endereço R4+2.

- **Instruções de dois operandos com modo de endereçamento As=01b e Ad=0b**

Para simular este tipo de instrução, foi feita a operação `ADD.W 2(R4),R5`, com o registrador R4 carregado previamente com o endereço 0200h, o registrado R5 carregado com o valor 0300h e a posição 0202h da memória de dados com o valor 0100h.

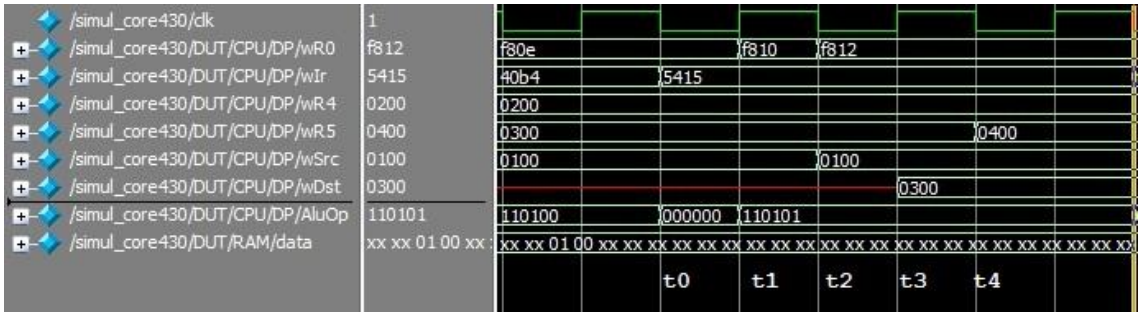

Figura 66 - Resultado da simulação de instruções de um operando com As=01 e Ad=0.

Em t_0, a instrução é carregada no registrador de instruções. Em t_1, o registrador PC é incrementado. Em t_2, o registrador SRC é carregado com o valor contido em 0202h. Em t_3, o operando contido em R5 é carregado no registrador DST. Em t_4, a operação é executada e o resultado é gravado em R5.

- **Instruções de dois operandos com modo de endereçamento As=01b e Ad=1b**

Para simular este tipo de instrução, foi feita a operação `ADD.W 2(R4),0(R5)`, com o registrador R4 carregado previamente com o endereço 0200h, o registrado R5 carregado com o endereço 0202h e a posição 0202h da memória de dados com o valor 0300h.

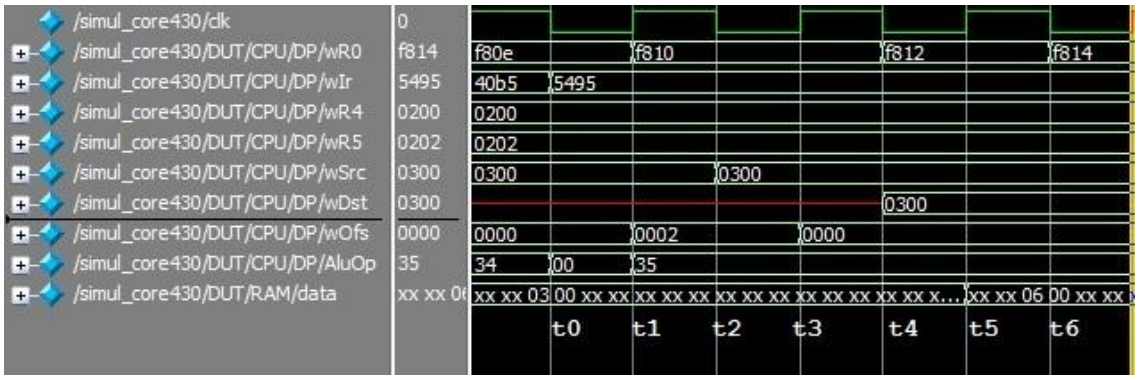

Figura 67 - Resultado da simulação de instruções de um operando com As=01 e Ad=1.

Em t_0, a instrução é carregada no registrador IR. Em t_1 a constante 0002h é carregada no registrador OFS/TEMP e o registrador PC é incrementado. Em t_2, o conteúdo apontado por

R4+2 é carregado no registrador SRC. Em t_3, a constante 0000h (apontada por PC+2) é carregada no registrador OFS/TEMP. Em t_4, o conteúdo apontado por R5+0 é carregado no registrador DST e PC+2 é gravado em PC. Em t_5, a operação é realizada e gravada no operando destino. Em t_6, PC novamente é incrementado, apontando para a próxima instrução.

- **Instruções de dois operandos com modo de endereçamento As=10b e Ad=0b**

Para simular este tipo de instrução, foi feita a operação `ADD.W @R4,R5` com o registrador R4 carregado previamente com o endereço 0200h, o registrado R5 carregado com o valor 0100h e a posição 0200h da memória de dados com o valor 0300h.

Figura 68 - Resultado da simulação de instruções de um operando com As=10 e Ad=0.

Em t_0, a instrução é carregada no registrador IR. Em t_1, PC é incrementado e o valor endereçado por R4 é carregado no registrador SRC. Em t_2, R5 é carregado no registrador DST. Em t_3, a operação é realizada e gravada no registrador R5.

- **Instruções de dois operandos com modo de endereçamento As=10b e Ad=1b**

Para simular este tipo de instrução, foi feita a operação `ADD.W @R4,2(R5)` com os registradores R4 e R5 carregados previamente com o endereço 0200h. O conteúdo do endereço 0200h foi inicializado com o valor 0100h e o conteúdo do endereço 0202h foi inicializado com o valor 0300h.

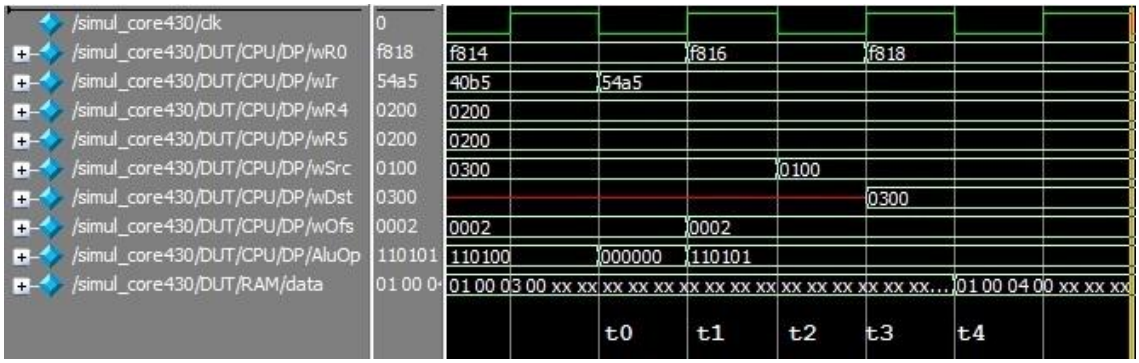

Figura 69 - Resultado da simulação de instruções de um operando com As=10 e Ad=1.

Em t_0, a instrução é carregada no registrador IR. Em t_1, PC é incrementado e o valor de *offset* 0002h é carregado no registrador OFS/TEMP. Em t_2, o valor endereçado por R4 é carregado no registrador SRC. Em t_3, o valor endereçado por R5+2 é carregado no registrador DST. Em t_4, a operação é executada e o valor é guardado no endereço de memória correspondente.

- **Instruções de dois operandos com modo de endereçamento As=11b e Ad=0b**

Para simular este tipo de instrução, foi feita a operação `ADD.W @R4+,R5` com o registradore R4 carregado previamente com o endereço 0200h. O conteúdo do endereço 0200h foi inicializado com o valor 0100h e o registrador R5 foi carregado com o valor 0300h.

Figura 70 - Resultado da simulação de instruções de um operando com As=11 e Ad=0.

Em t_0, a instrução é carregada no registrador de instruções. Em t_1, PC é incrementado e o registrador SRC é carregado com o valor endereçado por R4. Em t_2, R4 é incrementado e o registrador DST é carregado com o valor contido em R5. Em t_3, a operação é realizada e o resultado é gravado em R5.

- **Instruções de dois operandos com modo de endereçamento As=11b e Ad=1b**

Para simular este tipo de instrução, foi feita a operação ADD.W @R4+,2(R5) com os registradores R4 e R5 carregado previamente com o endereço 0200h. O conteúdo do endereço 0200h foi inicializado com o valor 0300h e o conteúdo do endereço 0202h foi inicializado com o valor 0100h.

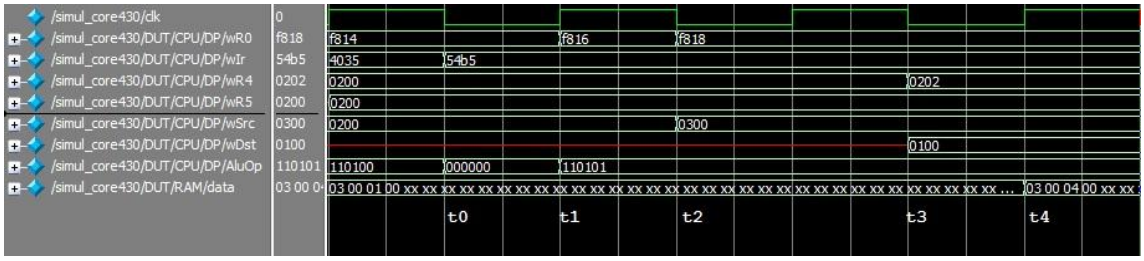

Figura 71 - Resultado da simulação de instruções de um operando com As=11 e Ad=1.

Em t_0, a instrução é carregada no registrador IR. Em t_1, PC é incrementado. Em t_2, o dado apontado pelo registrador R4 é carregado no registrador SRC. Em t_3, R4 é incrementado e o dado apontado por 2(R5) é carregado no registrador DST. Em t_4, a operação é executada e o resultado é armazenado em R5+2.

- **Instruções de salto**

Para simular as instruções de salto, o pequeno programa foi carregado na memória de programa.

Endereço	Conteúdo	Instrução
F800h	4034h	MOV.W #3,R4
F802h	0003h	
F804h (L1)	8314h	DEC.W R4
F806h	23FEh	JNZ L1
F808h	3FFFh	JMP $

Figura 72 - Programa usado na simulação das instruções de salto

O programa acima simplesmente inicializa o registrador R4 com o valor decimal 3 e o decrementa até 0. Então entra em loop infinito. Apesar de simples, esse programa mostra todas as possibilidades para instruções de salto: salto condicional com condição satisfeita, salto condicional com condição não-satisfeita e salto incondicional.

O resultado da simulação é mostrado pela figura 73.

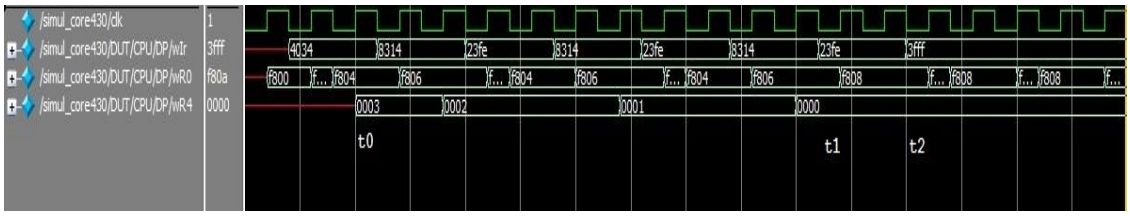

Figura 73 - Resultado da simulação das instruções de salto.

Pode-se observar na figura 73, entre os instantes t_0 e t_1, que o registrador PC é incrementado, mas logo em seguida é carregado com o endereço de L1. Entre t_1 e t_2 a condição de salto não é mais satisfeita e PC é incrementado para 0F808h. A partir de t_2, o programa entra em loop infinito, PC é incrementado e decrementado sucessivamente, ficando preso em 0F808h.

- **Instrução CALL**

Para simular a instrução *call*, o seguinte código-fonte foi usado.

Endereço	Conteúdo	Instrução
F800h	4031h	MOV.W #0280h,SP
F802h	0280h	
F804h	4304h	CLR.W R4
F806h	12B0h	CALL #L1
F808h	F810h	
F80Ah	3FFFh	JMP $
F80Ch	4303h	NOP
F80Eh	4303h	NOP
F810h (L1)	5314h	INC.W R4
F812h	4130h	RET

Figura 74 - Programa usado na simulação da instrução CALL.

O programa mostrado na figura 74 tem o propósito apenas de demonstrar o funcionamento da instrução *CALL*. Este programa simplesmente inicializa o registrador R4 com o valor 0. Em seguida chama uma sub-rotina para incrementá-lo em uma unidade. As instruções NOP foram usadas com o propósito de apenas criar uma distância entre o programa principal e a sub-rotina L1 para que a visualização do desvio de PC fosse mais clara.

O resultado da simulação da instrução CALL é mostrada na figura 75. Nela pode-se observar que durante a execução da instrução CALL, o registrador PC é atualizado de 0F80Ah para 0F810h, o endereço da sub-rotina. Também é mostrada a memória de programa depois da execução dessa instrução. Note que no último endereço da memória está contido o endereço

0F80Ah que corresponde ao endereço que PC deverá apontar quando a instrução RET é executada.

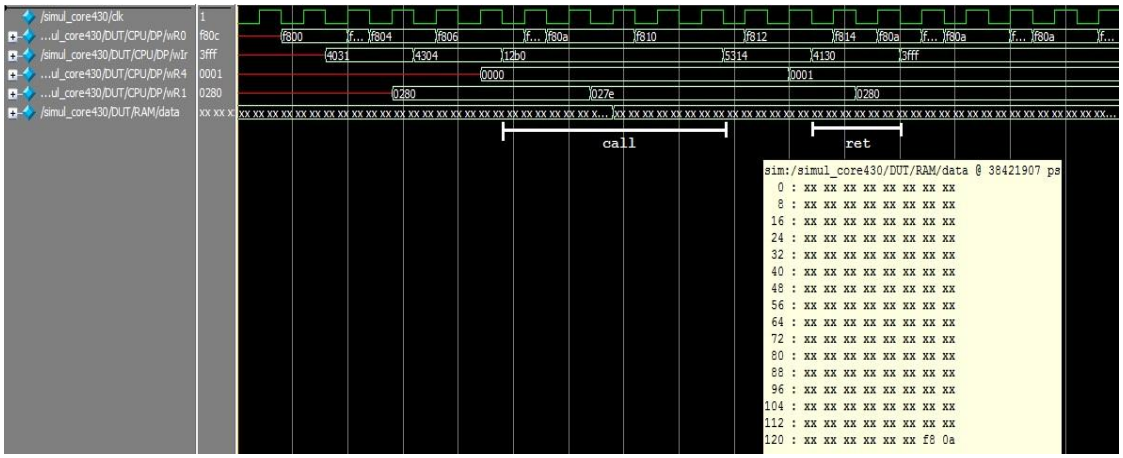

Figura 75 - Resultado da simulação da instrução CALL.

4.3. Port e interrupções

Para simular o funcionamento do *port* e também das interrupções, o seguinte programa foi utilizado.

Endereço	Conteúdo	Instrução	Comentário
F800h	4031h	MOV.W #0280h,SP	Inicializa SP
F802h	0280h		
F804h	40B2h	MOV.W #5A80h,&WDTCTL	Desliga o watchdog timer
F806h	5A80h		
F808h	0120h		
F80Ah	43D2h	MOV.B #1, &P1DIR	P1.0=saída, P1.1=entrada
F80Ch	0022h		
F80Eh	43E2h	MOV.B #2, &P1IE	Habilita interrupção em P1.1
F810h	0025h		
F812h	43C2h	MOV.B #0, &P1IES	Seta interrupção na borda de subida
F814h	0024h		
F816h	4232h	MOV.W #8, &ST	Habilita interrupções(GIE)
F818h	43C2h	MOV.W #0, &P1OUT	Reseta P1.0
F81Ah	0021h		
F81Ch	3FFFh	JMP $	Entra em loop infinito
F81Eh(intp1)	43C2h	MOV.B #0, &P1IFG	Reseta os flags de interrupções de P1
F820h	0023h		
F822h	43D2h	MOV.B #1, &P1OUT	Seta P1.0
F824h	0021h		
F826h	1300	RETI	Retorna da interrupção
FFE4h	F81Eh	Vetor de interrupção de P1	

Figura 76 - Código-fonte usado na simulação do port e das interrupções

Este programa simplesmente configura o pino P1.0 como saída, P1.1 como entrada, habilita as interrupções e entra em loop infinito aguardando uma transição positiva de P1.1. Quando a interrupção é acionada, a saída P1.0 é setada. O resultado da simulação é mostrado na figura 77. Para ajudar na interpretação do resultado, um flag interno chamado "interrupt_on" usado para debug foi mostrado nos sinais de saída.

Antes de t_0, o programa estava em loop aguardando a interrupção. Em t_0, P1.1 é setado e a interrupção é solicitada. Entre t_0 e t_1, a instrução sendo executada pela CPU é finalizada normalmente. Quando sua execução termina, assim que a borda "interrupt_on" sobe, significa que a CPU entrou no estado de interrupção. A partir daí, pode-se observar a sequência de eventos: R0/PC é empilhado, R2/SR é empilhado, R2/SR é carregado com 0000h, R0/PC é carregado com o endereço do vetor de interrupções do Port 1, e logo em seguida com o endereço da rotina de tratamento de interrupções (F81Eh) que estava contido no vetor.

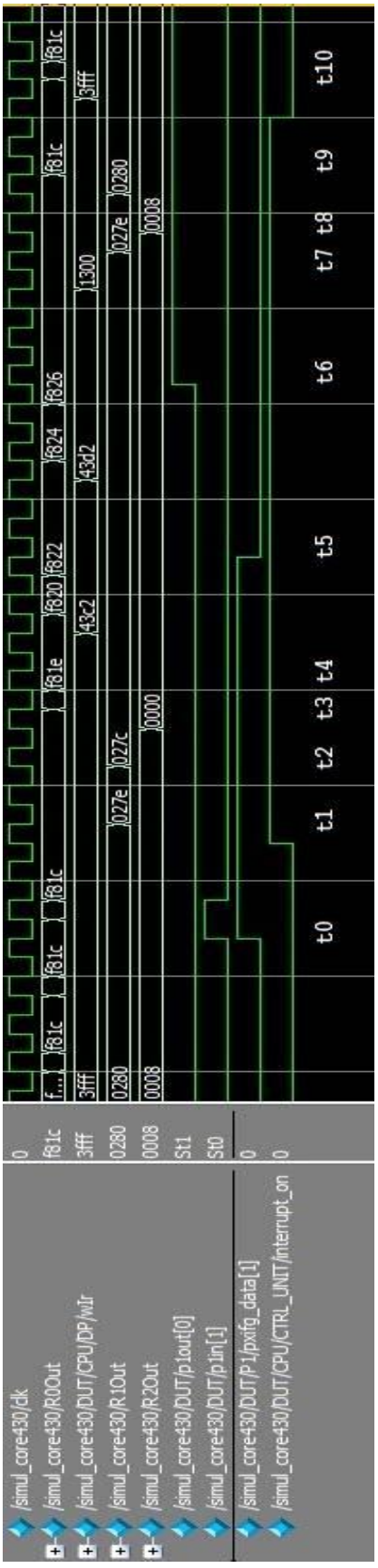

Figura 77 - Simulação do port/interrupção.

A partir daí, a CPU retorna ao estado de execução de instruções e continua executando as instruções normalmente: em t_5, o flag de interrupção é resetado; em t_6, o bit P1.0 é setado até que em t_7, a instrução RETI começa a ser executada. Então, em t_8 o registrador R2/SR é desempilhado da pilha. Em t_9, o registrator R0/PC também é desempilhado, encerrando o ciclo de interrupções. A partir de t_{10}, o programa volta ao loop que estava antes da solicitação de interrupção e segue normalmente com sua execução.

4.4. Módulo básico de clock

- **Divisor de clock**

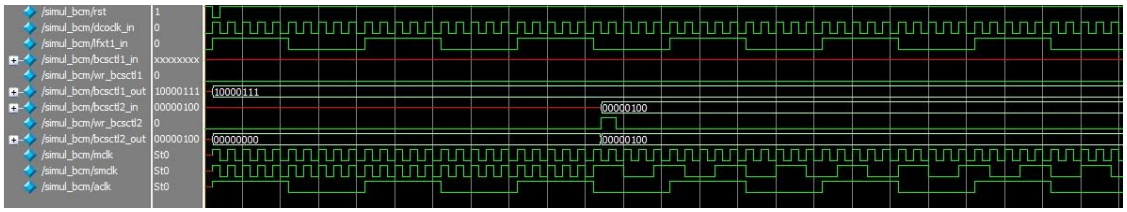

Figura 78 - Resultado da simulação do módulo básico de clock – Divisor.

Neste exemplo pode-se observar o funcionamento dos divisores de *clock*. Note que em determinado instante os bits DIVSx do registrador BCSCTL2 foram configurados com o valor 10b e imediatamente o sinal SMCLK teve sua frequência dividida por 4.

- **Origem do sinal clock**

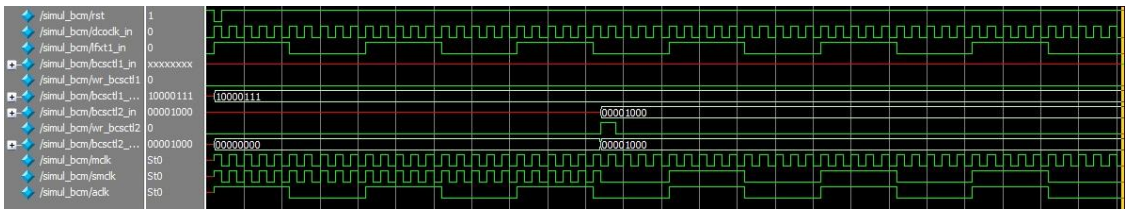

Figura 79 - Resultado da simulação do módulo básico de clock - Origem do sinal.

Neste exemplo pode-se observar como a origem do sinal de *clock* pode ser escolhida. Note que inicialmente o sinal SMCLK possui a frequência do sinal DCOCLK, então em determinado instante o bit SELS do registrador BCSCTL2 é setado, fazendo com que a origem do sinal passe a ser LFXT1.

4.5. Watchdog Timer

- **Dispositivo configurado como watchdog**

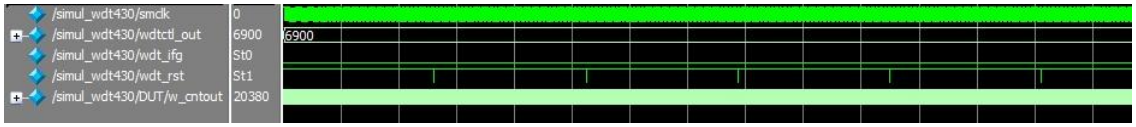

Figura 80 - Simulação do watchdog timer configurado como watchdog.

Figura 81 - Detalhe da simulação do watchdog timer configurado como watchdog.

A figura 80 mostra o que acontece quando o registrador WDTCTL não é configurado pelo software. O contador conta 32.768 pulsos de *clock* e então gera um pulso negativo de *reset*, reiniciando a CPU e os outros periféricos do microcontrolador. A figura 81 mostra em detalhes um desses pulsos.

- **Dispositivo configurado como timer**

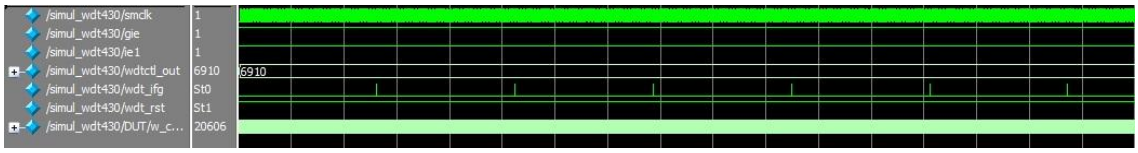

Figura 82 - Simulação do watchdog timer configurado como timer.

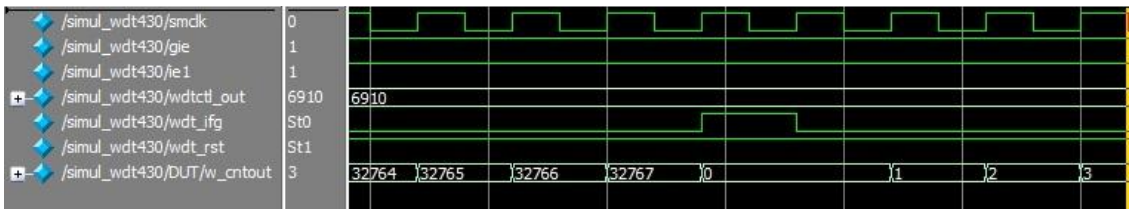

Figura 83 - Detalhe da simulação do watchdog timer configurado como timer.

A figura 82 mostra o comportamento do dispositivo quando configurado na função de temporizador, contando 32.768 pulsos de *clock*. Note que ao contrário do caso anterior, quando a contagem é alcançada, o dispositivo não gera pulsos negativos de *reset*, mas sim pulsos positivos de interrupção. A figura 83 mostra em detalhes o que acontece quando o contador atinge sua contagem máxima.

- **Violação do registrador WDTCTL**

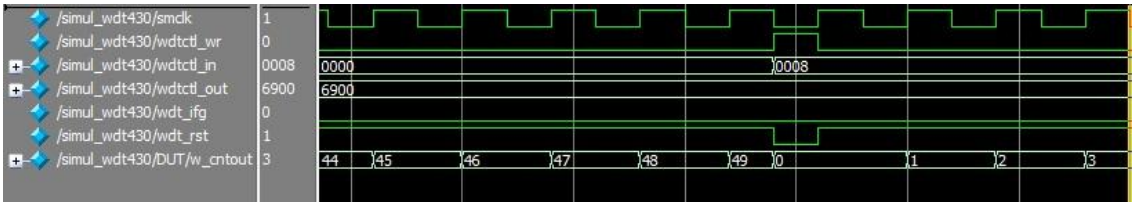

Figura 84 - Violação do registrador WDTCTL.

A figura 84 mostra o que acontece quando há uma tentativa de escrita no registrador WDTCTL sem a senha 5Ah no byte mais significativo. Note que no instante em que a escrita ocorre, imediatamente um sinal de *reset* é gerado.

4.6. Resultados das simulações do Timer A

Nesta seção são mostrados alguns resultados de simulações com o periférico *Timer A*. Serão mostradas simulações em seus diferentes modos de contagem e funcionamento.

- **Modo de comparação com contagem crescente**

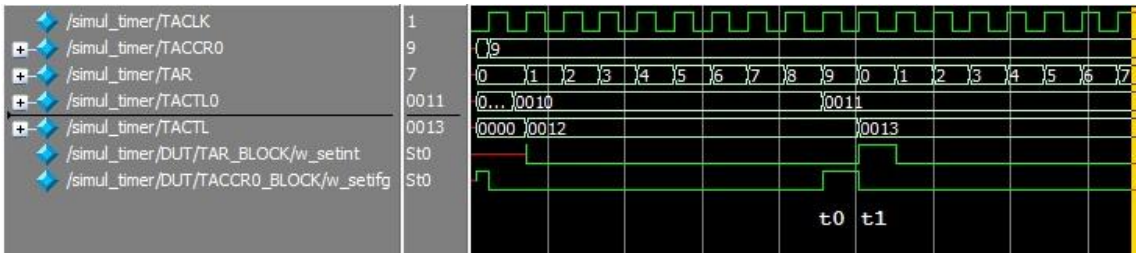

Figura 85 - Resultado da simulação do timer A em modo de comparação com contagem crescente.

Por meio da figura 85, pode-se observar o funcionamento do circuito do *timer A* implementado. Nesta simulação o registrador TACCR0 foi carregado com o valor 9. Neste modo de contagem, o registrador TAR é incrementado até o valor de TACCR0, então a contagem é reiniciada.

Note que em t_0, TAR = TACCR0. Nesse instante a interrupção do bloco de captura/comparação é setada, como pode ser visto pelo sinal interno "w_setifg" e pela mudança no valor do registrador TACTL0, onde o bit TACTL0[0] que corresponde a CCIFG é setado.

Em t_1, o valor do registrador TAR é resetado, habilitando a interrupção do bloco contador, mostrada pelo sinal "w_setint" e pela mudança no valor do bit 0 (TAIFG) registrador TACTL.

- **Modo de comparação com contagem contínua**

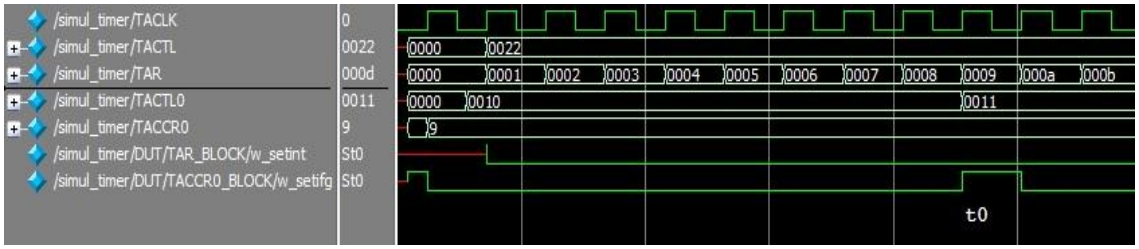

Figura 86 - Resultado da simulação do timer A em modo de comparação com contagem contínua (1).

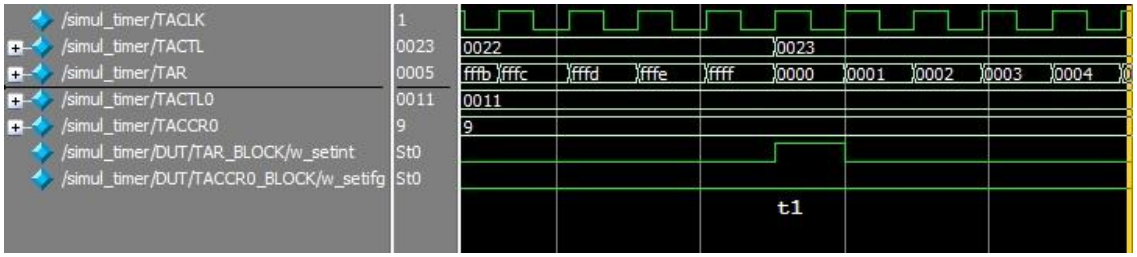

Figura 87 - Resultado da simulação do timer A em modo de comparação com contagem contínua (2).

Como pode ser observado pelas figuras 86 e 87, o funcionamento do timer em modo de contagem contínua é semelhante ao modo de contagem crescente, exceto que neste caso a contagem é feita continuamente até que ocorra *overflow* do contador.

Em t_0, pode ser observado que quando a contagem se torna igual ao valor contido em TACCR0, o flag CCIFG (TACTL0[0]) é setado. De forma semelhante, em t_1, pode ser observado o que o flag TAIFG (TACTL[0]) é setado quando ocorre *overflow* do registrador TAR.

- **Modo de comparação com contagem crescente/decrescente**

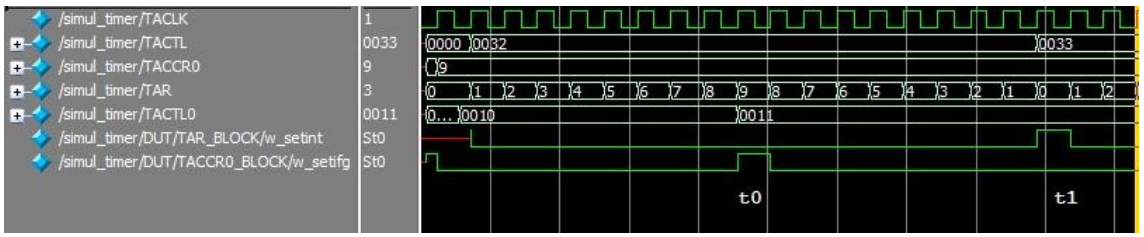

Figura 88 - Resultado da simulação do timer A em modo de comparação com contagem crescente/decrescente.

A figura 88, mostra o funcionamento do timer em modo de comparação em contagem crescente/decrescente. Nela pode ser observado que o registrador TAR é incrementado até alcançar o valor contido em TACCR0 e então passa a ser decrementado até atingir o valor 0, quando passa a ser incrementado novamente.

Em t_0, pode-se observar o *flag* CCIFG (TACTL0[0]) sendo setado quando a contagem atinge seu valor máximo. Da mesma forma, em t_1, pode-se observar o *flag* TAIFG (TACTL[0]) sendo setado quando a contagem se torna igual a 0.

- **Modo de captura**

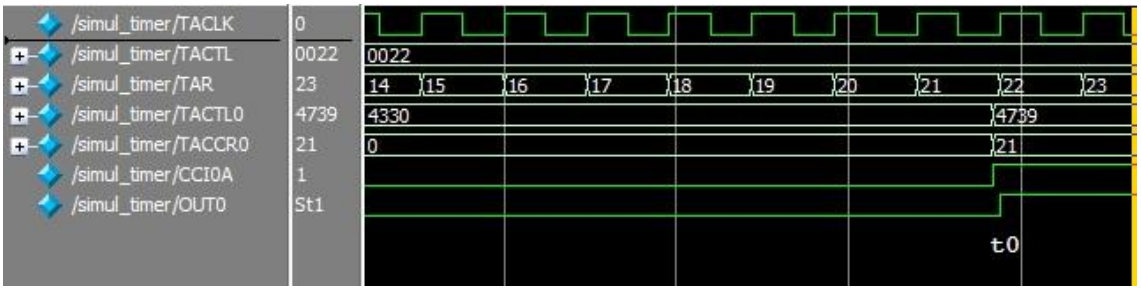

Figura 89 - Resultado da simulação do timer A em modo de captura.

O resultado da simulação do funcionamento do *timer A* em modo de captura pode ser observado pela figura 89. Neste exemplo, o módulo de captura/comparação foi configurado para fazer a captura na transição positiva do sinal externo CCI0A e a saída OUT0 foi configurada para operar no modo *Set/Reset*. As interrupções também foram habilitadas.

Pode-se observar, em t_0, que quando o sinal CCI0A passa de 0 para 1, imediatamente o valor do contador TAR é "capturado" em TACCR0 e o bit CCIFG (TACTL0[0]) é setado, solicitando interrupção. Em seguida, a saída OUT0 também é setada.

Observando a figura 89 com atenção é possível notar um pequeno *delay* entre a captura e o momento em que OUT0 é setado. Isso ocorre porque a captura pode ser feita a qualquer instante, porém o sinal OUT0 é sincronizado com a borda positiva do sinal de *clock* usado pelo timer.

4.7. Discussão dos Resultados

Os resultados mostrados neste capítulo, obtidos por meio das simulações realizadas com o *ModelSim*, se mostraram condizentes com o que ocorre no MSP430.

Embora seja impraticável simular todas as situações possíveis do microcontrolador, as simulações feitas cobrem de forma geral todas as aplicações mais comuns do microcontrolador.

As simulações da unidade lógica e aritmética (ULA) mostraram que todas as operações realizadas pelo MSP430 foram implementadas com sucesso.

As simulações feitas com a CPU mostram que esta é capaz de executar as instruções em todos os modos de endereçamento possíveis. Embora as simulações tenham sido feitas todas com instruções ADD e SWPB, os resultados obtidos podem ser generalizados para todas as outras instruções, pois a única diferença entre a execução dessas instruções e do restante é a operação carregada na ULA.

As simulações feitas com o módulo básico de *clock* mostram seu funcionamento compatível com o esperado, dentro das funções possíveis de ser implementadas em FPGA. Apesar de não contar com o oscilador interno DCO, isso não representa uma grande desvantagem para o projeto, pois muitas das aplicações de microcontroladores não requerem alterações na frequência do *clock* em tempo de execução.

As simulações feitas com o *watchdog timer* mostraram com sucesso o funcionamento em seus dois modos de operação: como temporizador de intervalos e como cão de guarda. Em ambos os casos os resultados satisfizeram as expectativas

As simulações feitas com o Timer A também se mostraram condizentes com o que ocorre no MSP430.

Os resultados obtidos através de todas as simulações citadas acima mostram que as implementações realizadas foram feitas corretamente e o projeto encontra-se pronto para a etapa de implementação física.

5. CONCLUSÕES

5.1 Contribuições

Este trabalho abordou o desenvolvimento em linguagem Verilog de um *core* compatível com o MSP430 para FPGAs visando unir todas as vantagens deste microcontrolador à flexibilidade das FPGAs. Os resultados obtidos por meio das simulações feitas mostraram que o projeto foi desenvolvido com sucesso, é funcional e está pronto para avançar para o próximo nível, a implementação física do *hardware* desenvolvido.

Por se tratar de uma implementação para FPGAs, algumas características analógicas do microcontrolador não puderam ser implementadas, como o oscilador analógico DCO do módulo básico de *clock*. Apesar de isso representar uma limitação, ela não é tão grande, pois muitos dos projetos envolvendo microcontroladores funcionam com *clock* constante.

No âmbito pessoal, este projeto contribuiu para que o autor aprofundasse seus conhecimentos na área de projeto e desenvolvimento de *hardware* digital. Além da experiência em desenvolvimento adquirida, foram incorporadas ao conhecimento do autor, uma nova linguagem de descrição de *hardware*, o Verilog, e um novo microcontrolador, o MSP430, não estudados durante o curso de graduação.

5.2 Trabalhos Futuros

Ao completar este projeto, ficam para trabalhos futuros a implementação física do hardware desenvolvido nos diferentes dispositivos FPGA disponíveis no mercado, além da implementação de outros periféricos que não foram tratados neste trabalho bem como ferramentas de auxílio ao desenvolvimento como *hardware* e *software* de *debug*.

REFERÊNCIAS

ALTERA. Cyclone II Device Handbook, Volume 1. 2008.

BARR, M. Programmable Logic: What's it to Ya? Embedded Systems Programming. Junho 1999, pp. 75-84.

DAVIES, J. MSP430 Microcontroller Basics. Newnes, 2008.

MSP430 Microarchitectural Simulator. Disponível em: http://ece124web.groups.et.byu.net/labs/L06-archsim/archsim.html

NAVABI, Z. Embedded Core Design with FPGAs. McGraw-Hill, 2007.

OpenMSP430. Disponível em: http://opencores.org/project,openmsp430

Synthesizable MSP430. Disponível em: http://savannah.nongnu.org/cvs/?group=s430

TEXAS INSTRUMENTS. MSP430x2xx Family User's Guide. 2012.

TEXAS INSTRUMENTS. MSP430G2x31Datasheet. 2011.

TOCCI J.R.; WIDMER S. N.; MOSS L. G. Sistemas Digitais Princípios e Aplicações. Pearson Prentice Hall, 2008.