DOCUMENTAÇÃO DO MICROCONTROLADOR R8_uC
PROJETO DE PROCESSADORES - ELC 1094 - Prof. Carara (carara@ufsm.br)

Carlos Gabriel de Araujo Gewehr (*gabriel.gewehr@ecomp.ufsm.br*)
Emilio Garcez Ferreira (*emilio.ferreira@ecomp.ufsm.br*)

1. OVERVIEW

O microcontrolador R8_uC, desenvolvido durante a disciplina de Projeto de Processadores ao longo do primeiro semestre de 2019 consiste em um processador multiciclo R8[1] modificado de modo a suportar o controle e acesso a alguns periféricos, descritos em detalhe na seção 3 deste documento, assim como uma base de software que explora as funcionalidades implementadas, tal como o tratamento vetorizado de interrupções, assim como o suporte a traps, e acesso aos periféricos por meio de *syscalls* (Chamadas de sistema).

O processador R8 modificado funciona de forma quase idêntica ao processador R8 original: Se mantém a arquitetura load/store, com 16 registradores de propósito geral, instruções de tamanho fixo, executadas em ou 3 ou 4 ciclos, flags geradas pela ALU de resultado *negativo* (n), *zero* (z), *carry* (c) e *overflow* (v). (As modificações feitas foram orientadas a adição de novas funcionalidades, não a reformulação de funcionalidades já existentes).

Os periféricos anexados ao microcontrolador podem ser utilizados a partir da escrita ou leitura em registradores contidos nos periféricos (ver Tabela 1) a partir de um barramento de dados compartilhado com a memória por simples instruções de acesso à memória (ou seja, não há instruções dedicadas para o acesso a periféricos), efetuadas em endereços únicos à cada periférico. (Em síntese, para utilização de um periférico deve ser executada uma instrução de acesso a memória na faixa de endereços do periférico desejado. Esse processo é exemplificado em grande detalhe na seção 5 deste documento).

O microcontrolador também contém uma memória RAM, onde estão contidos ambas instruções e dados, assim como uma memória ROM, contida nesta um programa responsável pelo carregamento (na memória RAM) do software a ser executado no microcontrolador a partir de dados recebidos através do periférico UART RX.

É possível ter uma visão mais clara dos elementos contidos no microcontrolador desenvolvido a partir do diagrama de blocos da figura abaixo:

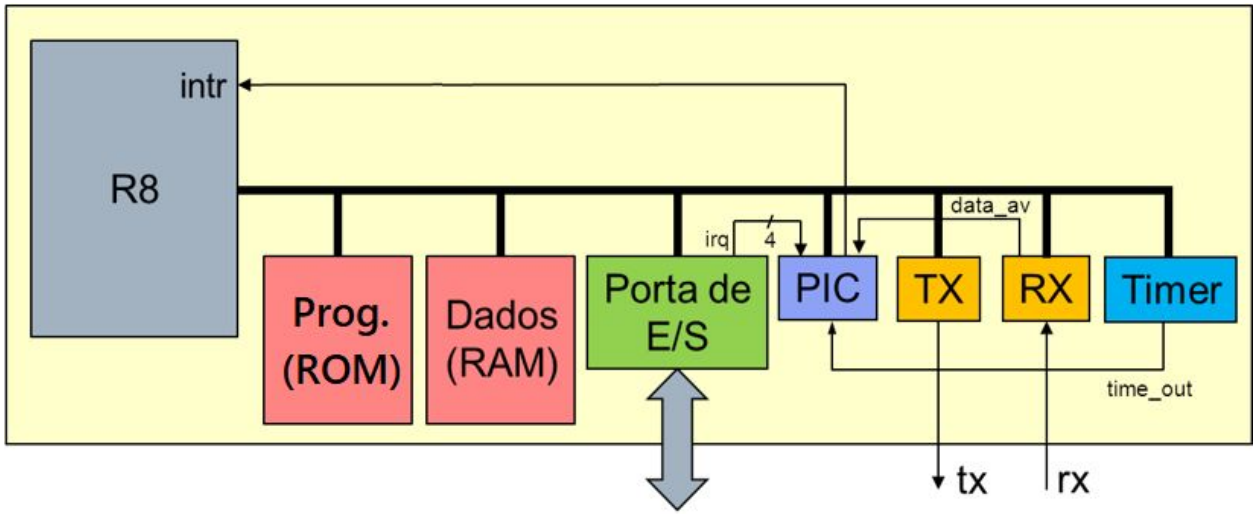

Figura 1 - Diagrama mostrando processador e periféricos do R8_uC

É possível observar mais detalhes sobre as interconexões dos elementos do microcontrolador a partir da figura abaixo, que mostra a divisão do projeto de acordo com os arquivos VHDL disponibilizados (ver seção 5), e sua prototipação na placa Nexys 3.

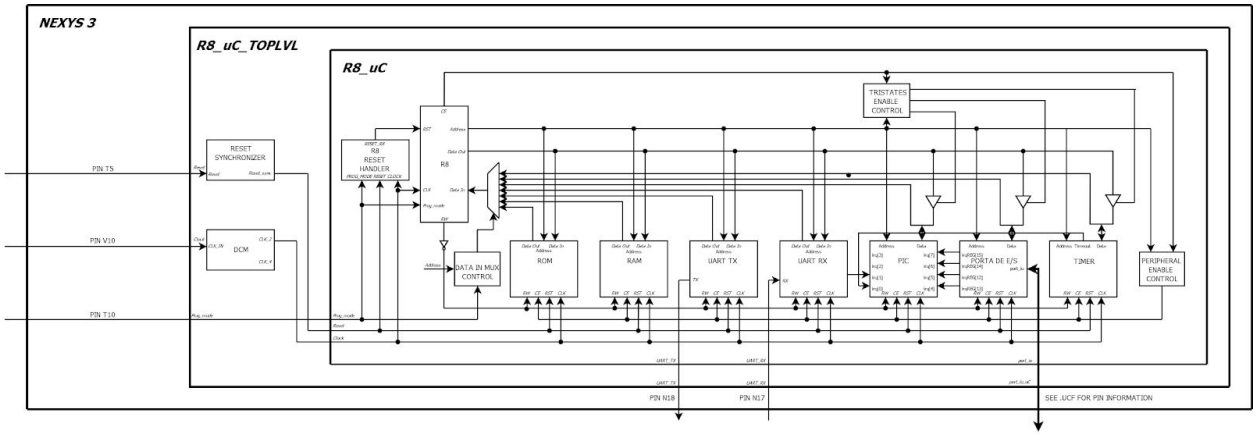

Figura 2 - Diagrama aprofundado do R8_uC, mostrando detalhes da implementação

Abaixo estão apresentados os registradores contidos nos periféricos, e os endereços de memória onde deve ser executadas operações de escrita ou leitura a fim de manipulá-los. (Na seção 3 deste documento também são apresentados exemplos demonstrando o uso e alguns detalhes notáveis de implementação de cada periférico presente no microcontrolador, assim como disponibilizados em anexo, os códigos fonte dos programas exemplo presentes na seção 3).

Periferico	Registrador	Endereço
Porta de E/S	PortData	0x8000
Porta de E/S	PortConfig	0x8001
Porta de E/S	PortEnable	0x8002
Porta de E/S	irqtEnable	0x8003
PIC	irqID	0x80F0
PIC	intACK	0x80F1
PIC	Mask	0x80F2
PIC	irqREG	0x80F3
UART TX	TX_DATA	0x8080
UART TX	RATE_FREQ_BAUD	0x8081
UART TX	READY	0x8082
UART RX	RX DATA	0x80A0
UART RX	RATE_FREQ_BAUD	0x80A1
TIMER	Counter	0x80B0

Tabela 1 - Endereços dos registradores dos periféricos do microcontrolador

2. PROCESSADOR R8

Como dito na seção 1, o processador R8 contido no microcontrolador desenvolvido em essência pouco difere do processador R8, originalmente desenvolvido por Moraes e Calazans (documentação disponível junto a esse documento), usado como base no desenvolvimento deste projeto, novamente, visando a adição de funcionalidades. Além de manter as características descritas no começo da seção 1, são mantidos os registradores de propósito específico *regPC* (Program Counter, contendo o endereço de memória onde está localizada a próxima instrução a ser executada), *regSP* (Stack Pointer, contendo o endereço de memória onde a próxima operação de *push* na pilha será executada), *regIR* (Instruction Register, contendo a instrução sendo executada), assim como o banco de registradores de 16 registradores de propósito geral (Os registradores de propósito específico estão documentados em maior detalhe ao final desta seção). As diferenças entre os dois processadores estão listadas abaixo:

- Enquanto o processador original está descrito em VHDL de forma estrutural, o processador modificado está descrito também em VHDL, porém de forma comportamental, devido a facilidade de adição de funcionalidades e teste das mesmas. (Devido ao caráter didático do projeto, uma descrição comportamental se mostra superior, graças à clareza almejada na descrição da arquitetura do processador, e como já dito, a facilidade de implementação de novas instruções e funcionalidades, e mais rápida verificação da corretude das mesmas). A decodificação das instruções acontece de forma combinacional, sua execução dentro de uma máquina de estados, descrita (em VHDL) através de um *process*, sensível a borda de subida do sinal de *clock* e o sinal de *reset*, enquanto que os sinais da ALU e de controle de acesso à memória são gerados combinacionalmente, também de forma comportamental, de acordo com o estado da máquina de estados que rege a execução da instrução atual.

- Adição de multiplicador e divisor comportamentais, assim como registradores de propósito específico: *regHIGH*, que contém a parte alta da multiplicação ou o resto da divisão; e *regLOW*, que contém a parte baixa da multiplicação ou o quociente da divisão, de acordo com a instrução emitida. Para a execução dessas operações, foram implementadas novas instruções: MUL e DIV, que atualizam os valores de *regHIGH* e *regLOW*, como descrito acima; e MFH e MFL efetuando a transferência do dado contido em, respectivamente, *regHIGH* ou *regLOW* para um dos registradores de propósito geral.

- Adição de um registrador de propósito específico, *regFLAGS*, contendo as flags geradas pela ALU, e duas instruções para salvá-lo/restaurá-lo na pilha, PUSHF, e POPF, respectivamente, salvando na pilha o conteúdo de *regFLAGS*, e carregando em *regFLAGS* o dado contido no topo da pilha. (Deve-se lembrar que o endereço da próxima posição da pilha, onde será salvo o conteúdo do registrador desejado em uma operação de *push*, está contido no registrador de propósito específico *regSP*).

- Adição de suporte a interrupções, através da verificação do novo sinal de entrada (gerado fora do processador) *irq* no estado de *fetch* (busca da próxima instrução a ser executada na memória, de acordo com o endereço contido em *regPC*). Se o sinal *irq* estiver ativo, e processador não estiver tratando outra interrupção (sendo isso sinalizado por um novo sinal: *interruptFlag*), o fluxo de decodificação e execução da instrução atual é desviado para outro estado na máquina de estados do processador, *Sitr*, onde o valor atual de *regPC* é salvo na pilha e no novo registrador de propósito específico *regITR*. Feito isso, ainda no mesmo estado, o registrador *regPC* recebe o valor contido em um novo registrador de propósito específico, *regISRA*, e ativa o sinal *interruptFlag*. O registrador *regISRA*, (onde ISRA é uma sigla para *Interrupt Service Routine Address*) contém o endereço de memória onde está localizada a primeira instrução de uma subrotina de tratamento de interrupções (O registrador regISRA pode ser carregado a partir de um registrador de propósito genérico através de uma nova instrução, LDISRA). Para o retorno ao fluxo de execução normal do programa (após tratada a interrupção) deve ser executada a nova instrução RTI (*Return From Interruption*) na subrotina de tratamento de interrupção, onde é restaurado o valor de *regPC*, e desativado o sinal *interruptFlag*. Deve-se notar que outras operações são realizadas durante a execução da instrução RTI, estas relativas ao tratamento de *traps*, conforme descrito no próximo item. (OBS: O processo de decisão feito ao optar por salvar o conteúdo de *regPC* tanto em um registrador de propósito específico quanto na pilha é comentado sobre ao final da seção 2).

- Adição de suporte a tratamento de traps, feito de forma similar ao tratamento de interrupções, tal como descrito no item acima. Analogamente ao registrador *regISRA* (*Traps Service Routine Address*) e a instrução LDISRA, foi adicionado um novo registrador de propósito específico, *regTSRA* e uma nova instrução LDTSRA, que torna possível o carregamento de *regTSRA* a partir de um registrador de propósito genérico. Assim como o tratamento de interrupções, o tratamento de traps começa no estado de *fetch*. Se o novo sinal *newTrapFlag* estiver ativo, o novo estado da máquina de estados será *Strap*, onde o valor atual de *regPC* é salvo no registrador *regITR* e na pilha, o sinal *newTrapFlag* é desativado, o sinal *interruptFlag* é ativado, o novo sinal *trapCount* é incrementado, e o registrador *regPC* recebe o valor contido em *regTSRA*. Deve-se notar que a verificação quanto a existência de uma trap a ser tratada ocorre antes da verificação de uma interrupção pendente, de modo a ser possível o tratamento de traps dentro da rotina de tratamento de interrupção. Desse modo, também deve-se notar que o sinal *interruptFlag* é feito ativo dentro do estado *Strap* de forma a evitar que o tratamento de uma interrupção ocorra durante o tratamento de uma trap, forçando o tratamento de traps a ser efetuado como uma operação contínua. O sinal *interruptFlag* será desativado na instrução RTI, conforme descrito no item acima, somente se todas as traps pendentes foram tratadas, ou seja, quando o sinal *trapCount* for igual a zero. O sinal *trapCount* será decrementado, se não for igual a zero (se trapCount for igual a

zero, a chamada de RTI é relativa ao final do tratamento de uma interrupção, não de uma *trap*). Foram também adicionadas duas novas instruções, MFC (Move From Cause), que de forma análoga as instruções MFH e MFL, permite o carregamento de um dos registradores de propósito geral com o conteúdo do novo registrador *regCAUSE*, este que contém o número identificador sobre a nova trap gerada, e MFT, que carrega um dos registradores de propósito geral com o valor de *regPC* salvo em *regITR*, ou seja, a instrução que gerou a trap a ser tratada. Finalmente, deve-se notar que uma instrução que gera uma trap não tem sua execução efetivamente completa, por exemplo, uma divisão por zero, são atualizará os valores de *regHIGH* e *regLOW*.

Abaixo está disponível uma tabela com todas as instruções do processador R8 modificado, sua codificação, e a ação a qual efetuam, assim como uma tabelas com os números identificadores de *syscalls*, bem como uma tabela com as *traps* e seus números identificadores:

Instrução	Formato				Ação ; *flags*
	15-12	11-8	7-4	3-0	
ADD Rt, Rs1, Rs2	0	Rt	Rs1	Rs2	Rt <= Rs1 + Rs2 (n, z, c, v)
SUB Rt, Rs1, Rs2	1	Rt	Rs1	Rs2	Rt <= Rs1 - Rs2 ; (n, z, c, v)
AND Rt, Rs1, Rs2	2	Rt	Rs1	Rs2	Rt <= Rs1 and Rs2 ; (n, z)
OR Rt, Rs1, Rs2	3	Rt	Rs1	Rs2	Rt <= Rs1 or Rs2 ; (n, z)
XOR Rt, Rs1, Rs2	4	Rt	Rs1	Rs2	Rt <= Rs1 xor Rs2 ; (n, z)
ADDI Rt, #cte8	5	Rt	cte8		Rt <= Rt + cte8 ; (n, z, c, v)
SUBI Rt, #cte8	6	Rt	cte8		Rt <= Rt - cte8 ; (n, z, c, v)
LDL Rt, #cte8	7	Rt	cte8		Rt <= Rt(15:8) & cte8
LDH Rt, #cte8	8	Rt	cte8		Rt <= cte8 & Rt(7:0)
LD Rt, Rs1, Rs2	9	Rt	Rs1	Rs2	Rt <= MEM[Rs1 + Rs2]
ST Rt, Rs1, Rs2	A	Rt	Rs1	Rs2	MEM[Rs1 + Rs2] <= Rt
SL0 Rt, Rs1	B	Rt	Rs1	0	Rt <= Rs1[14:0] & 0 ; (n, z)
SL1 Rt, Rs1	B	Rt	Rs1	1	Rt <= Rs1[14:0] & 1 ; (n, z)
SR0 Rt, Rs1	B	Rt	Rs1	2	Rt <= 0 & Rs1 [15:1] ; (n, z)
SR1 Rt, Rs1	B	Rt	Rs1	3	Rt <= 1 & Rs1 [15:1] ; (n, z)
NOT Rt, Rs1	B	Rt	Rs1	4	Rt <= not Rs1 ; (n, z)
NOP	B	-	-	5	-

HALT	B	-	-	6	Espera interrupção infinitamente
LDSP Rs2	B	-	Rs2	7	regSP <= Rs2
RTS	B	-	-	8	regPC <= MEM[regSP+1], regSP++
POP	B	Rt	-	9	Rt <= MEM[regSP+1], regSP++
PUSH	B	Rt	-	A	MEM[regSP] <= Rt, regSP--
MUL Rs1, Rs2	B	Rs1	Rs2	B	regHIGH <= Rs1 * Rs2 [32:16] regLOW<= Rs1 * Rs2 [15:0]
DIV Rs1, Rs2	B	Rs1	Rs2	C	regHIGH <= Rs1 % Rs2, regLOW<= Rs1 / Rs2
MFH Rt	B	Rt	-	D	Rs1 <= regHIGH
MFL Rt	B	Rt	-	E	Rs1 <= regLOW
LDISRA Rs1	B	-	Rs1	F	regISRA <= Rs1
JMPR Rs1	C	0	Rs1	0	regPC <= regPC + Rs1
JMPNR Rs1	C	0	Rs1	1	regPC <= regPC + Rs1 ; (n=1)
JMPZR Rs1	C	0	Rs1	2	regPC <= regPC + Rs1 ; (z=1)
JMPCR Rs1	C	0	Rs1	3	regPC <= regPC + Rs1 ; (c=1)
JMPVR Rs1	C	0	Rs1	4	regPC <= regPC + Rs1 ; (v=1)
JMP Rs1	C	0	Rs1	5	regPC <= Rs1
JMPN Rs1	C	0	Rs1	6	regPC <= Rs1 ; (n=1)
JMPZ Rs1	C	0	Rs1	7	regPC <= Rs1 ; (z=1)
JMPC Rs1	C	0	Rs1	8	regPC <= Rs1 ; (c=1)
JMPV Rs1	C	0	Rs1	9	regPC <= Rs1 ; (v=1)
JSRR Rs1	C	0	Rs1	A	MEM[regSP] <= regPC, regSP--, regPC <= regPC+ Rs1
JSR Rs1	C	0	Rs1	B	MEM[regSP] <= regPC, regSP--, regPC <= Rs1
PUSHF	C	0	-	C	MEM[regSP] <= regFLAGS, regSP--

POPF	C	0	-	D	regFLAGS <= MEM[regSP+1], regSP++
RTI	C	0	-	E	regPC <= MEM[regSP+1], regSP++, itrFlag <= 0, tCount--
SYSCALL	C	1	-	-	regPC <= regTSRA, regCAUSE <= 8
LDTSRA Rs1	C	2	Rs1	-	regTSRA <= Rs1
MFC Rs1	C	3	Rs1	-	Rs1 <= regCAUSE
MFT Rs1	C	4	Rs1	-	Rs1 <= regITR
JMPD #desloc	D	0	Deslocamento (10 bits)		regPC <= regPC + desloc
JMPND #desloc	E	0	Deslocamento (10 bits)		regPC <= regPC + desloc, (n=1)
JMPZD #desloc	E	1	Deslocamento (10 bits)		regPC <= regPC + desloc, (z=1)
JMPCD #desloc	E	2	Deslocamento (10 bits)		regPC <= regPC + desloc, (c=1)
JMPVD #desloc	E	3	Deslocamento (10 bits)		regPC <= regPC + desloc, (v=1)
JSRD #desloc	F	Deslocamento (12 bits)			regPC <= regPC + desloc, MEM[regSP] <= regPC, regSP--

Tabela 2 - Instruções, suas codificações, e ações que executam

Onde,

& : concatenação de vetores de bits
<= : atribuição de valor a registrador ou posição de memória
MEM[x] : conteúdo de posição de memória cujo endereço é “x“
Rt : regTarget
Rs1 : regSource1
Rs2 : regSource2
n : flag da ALU de resultado negativo
z : flag da ALU de resultado igual a zero
c : flag da ALU de carry
v : flag da ALU de overflow
(x=1) : necessita flag “x” (n, z, c, v) ativa para execução da instrução
itrFlag : sinal interruptionFlag
tCount : sinal trapCount

A tabela abaixo apresenta os números identificadores de cada trap ao lado de qual *trap* pertencem:

Trap	Numero identificador (em *regCAUSE*)
Acesso à primeira posição de memória RAM	0
Instrução Invalida	1
Syscall	8
Overflow em complemento de 2	12
Divisão por zero	15

Tabela 3 - Traps e seus número identificadores

Deve-se notar que a trap de número 0, Acesso a primeira posição de memória RAM, não acontece enquanto o processador estiver em modo de programação (gravando dados recebidos por UART RX na memória RAM, através de um programa cujas instruções a ser executadas se encontram na memória ROM, conforme descrito na seção 1), a fim de poder gravar na memória RAM alguma instrução ou dado na posição de memória 0x0000, a ser utilizado quando o processador não estiver em modo de programação.

Os novos registradores de propósito específico estão listados abaixo, bem como a necessidade de sua existência. Também estão exemplificadas as instruções necessárias para seu uso e manipulação:

regHIGH e **regLOW**: A fim de implementar instruções de multiplicação e divisão (MUL e DIV) fizeram-se necessários estes dois novos registradores, para armazenar o resultado da multiplicação entre o conteúdo de dois registradores de propósito geral (que, dependendo dos valores contidos nos registradores desejados, pode ser um valor somente representável com 32 bits, por isso a separação entre parte alta (*regHIGH*) e parte baixa (*regLOW*) do resultado em dois registradores de 16 bits) ou o quociente, em *regLOW*, e resto, em *regHIGH*, da divisão entre o conteúdo de dois registradores de propósito geral.

```
;   r1 <= 255
    ldh r1, #0
    ldl r1, #255

;   r2 <= 33
    ldh r2, #0
    ldl r2, #33

;   r1 * r2 = 8415 = 0x20DF (regHIGH <= 0x20, regLOW <= 0xDF)
    mul r1, r2

;   r11 <= 0x20
    mfh r11

;   r12 <= 0xDF
    mfl r12

;   Nesse ponto r11 contem a parte alta da multiplicação entre r1 e r2, e r12 a parte baixa
    nop

;   r1 / r2, quociente = 7, resto = 24
    div r1, r2

;   r11 <= 24
    mfh r11

;   r12 <= 7
    mfl r12

;   Nesse ponto r11 contem o resto da divisão entre r1 e r2, e r12 o quociente
    nop
```

Figura 3 - Uso dos registradores regHIGH e regLOW

regISRA: Este registrador tem em seu conteúdo o endereço da subrotina responsável por tratar interrupções (*Interruption Service Routine*). É possível o carregamento de dados nesse registrador através da instrução LDISRA (ver Tabela 2), que copia a informação contida em um dos 16 registradores de propósito geral e carrega-a em *regISRA*.

```
;   r1 <= &InterruptionServiceRoutine (ISR)
    ldh r1, #InterruptionServiceRoutine
    ldl r1, #InterruptionServiceRoutine

;   regISRA <= &InterruptionServiceRoutine
    ldisra r1

;   Nesse ponto regISRA contém o endereço da ISR
    nop
```

Figura 4 - Uso do registrador regISRA

regTSRA: Este registrador tem em seu conteúdo o endereço da subrotina responsável por tratar interrupções (*Traps Service Routine).* É possível o carregamento de dados nesse registrador através da instrução LDTSRA (ver Tabela 2), que copia a informação contida em um dos 16 registradores de propósito geral e carrega-a em *regTSRA*.

```
;   r1 <= &TrapsServiceRoutine (TSR)
    ldh r1, #TrapsServiceRoutine
    ldl r1, #TrapsServiceRoutine

;   regTSRA <= &TrapsServiceRoutine
    ldtsra r1

;   Nesse ponto regTSRA contém o endereço da TSR
    nop
```

Figura 5 - Uso do registrador regTSRA

regITR e **regCAUSE**: A necessidade dos registradores regITR e regCAUSE é devido a funcionalidade de tratamento de traps, como descrito ao começo desta seção. O registrador *regITR* contém o valor de *regPC* quando é detectada uma interrupção ou *trap*. Já o registrador *regCAUSE* contém o número identificador da última trap a ser gerada, conforme a Tabela 3.

```
;   r1 <= &TrapsServiceRoutine (TSR)
    ldh r1, #TrapsServiceRoutine
    ldl r1, #TrapsServiceRoutine

;   regTSRA <= &TrapsServiceRoutine
    ldtsra r1

;   r1 <= 0x7FFF = 32767 (Maior numero representavel em complemento de 2 com 16 bits)
    ldh r1, #7Fh
    ldl r1, #FFh

;   r1 <= 0x8000 = -32768 (Signed) (Overflow)
    addi r1, #1

;    No estado de busca dessa instrução será identificada uma trap de código 12 (Overflow)
;   em regCAUSE estará contido o valor 12 (ID da trap) e em regITR o endereço de memória da instrução addi + 1
;   Também deve-se notar que está instrução somente será executada após o tratamento da trap gerada
    nop

;   r2 <= Identificador da ultima trap gerada (12, Overflow)
    mfc r2

;   r3 <= Endereço da ultima instrução a gerar uma trap + 1 (&(addi r1, #1) + 1 = &(nop) )
    mft r3
```

Figura 6 - Uso dos registrados regCAUSE e regITR

Deve-se notar que o registrador *regITR* é redundante perante as funcionalidades almejadas, o endereço de memória de uma instrução que gera uma *trap* pode ser obtido da pilha, porém, sua existência é baseada na facilidade de obter essa informação em simulador HDL, facilitando o processo de *debug*. Se assim desejado, esse registrador pode ser removido do processador, assim como a instrução MFT, porém é necessário mudar a lógica de obtenção do endereço da instrução que gerou a última *trap*, dentro da TSR, em detrimento do uso da instrução MFT.

3. PERIFERICOS

Como visto na Figura 1, os periféricos disponíveis no microcontrolador são: Receptor e transmissor UART (com taxa Baud programável), Porta de E/S (com bits individualmente habilitáveis e configuráveis como Entrada ou Saída), Timer (em ciclos de clock), e PIC (Controlador de interrupções).

PIC: O PIC é um controlador de interrupções mascaráveis, que admite 8 entradas, de prioridades decrescentes, que, quando alguma dessas entradas está ativa (e não mascarada), interrompe o processador e grava no registrador *irqID* o identificador da entrada de mais alta prioridade ativa. As entradas de interrupções são mapeadas a bits do registrador *irqREG* (por exemplo, a entrada 5 do PIC é mapeada ao bit 5 de *irqREG*), que são verificadas em ordem crescente (de 0 até 7, ou seja, de maior prioridade até menor prioridade) pelo PIC. O mascaramento de entradas individuais se dá a partir do registrador Mask, cujos bits individuais pode desabilitar a entrada relativa a seu bit correspondente em *irqREG.* Por exemplo, se o bit 4 do registrador Mask for 0, e se a entrada de número 4 estiver ativa, apesar do o bit 4 do registrador *irqREG* ser 1, esse fato não implicará em uma interrupção, pois a verificação de uma interrupção pendente se dá a partir de uma operação *and* bit-a-bit entre os registradores *irqREG* e Mask. O bit menos significativo do resultado dessa operação que estiver ativo indica a interrupção de maior prioridade a ser gerada, que é identificada a partir da escrita em *irqID*, conforme descrito acima. Finalmente, um bit indicando uma interrupção pendente em *irqREG* continua ativo mesmo que sua entrada correspondente seja ativada e logo após desativada. Em outras palavras, o PIC é capaz de armazenar a informação sobre uma interrupção pendente a partir de somente um pulso da entrada, o que possibilita ao microcontrolador tratar uma interrupção de menor prioridade que é gerada enquanto está sendo tratada uma interrupção de maior prioridade. Um bit só é desativado em *irqREG* quando é efetuada uma escrita no registrador *intACK*, sinalizando que a interrupção cujo número identificador (e consequentemente, seu bit correspondente em *irqREG*) é escrito em *intACK,* indicando que a interrupção foi tratada pelo processador com sucesso. O endereço de memória para acesso a esses registradores pode ser verificado conforme a Tabela 1 ou a tabela abaixo:

Registrador	Endereço	Funcionalidade
irqID	0x80F0	Interrupção de maior prioridade a ser tratada
intACK	0x80F1	ID da última interrupção tratada pelo processador
Mask	0x80F2	Habilita individualmente entradas de possíveis interrupções
irqREG	0x80F3	Interrupções pendentes (não considera mascaramento)

Tabela 4 - Registradores do periférico PIC

Número da Entrada	Sinal da Entrada
0	Timer
1	UART RX
2	-
3	-
4	portIRQ[12]
5	portIRQ[13]
6	portIRQ[14]
7	portIRQ[15]

Tabela 5 - Sinais de entrada do PIC

```
;   r0 <= 0
    xor r0, r0, r0

;   r1 <= 0x00C0 = "00000000_11110000" (Habilita apenas interrupções geradas pela porta de E/S)
    ldh r1, #00h
    ldl r1, #C0h

;   r2 <= 0x80F2 = &Mask
    ldh r2, #80h
    ldl r2, #F2h

;   Mask <= 0x00C0
    st r1, r0, r2
```

Figura 7 - Uso do periférico PIC

Timer: Gera uma interrupção (entrada 0 no PIC) quando o conteúdo do registrador *Counter*, que é decrementado por uma unidade a cada ciclo de clock, chega a zero. Deve-se notar que esse periférico pode ser melhor utilizado juntamente ao *syscall* SetTimer, descrito em detalhe na seção 4 deste documento.

Registrador	Endereço	Funcionalidade
Counter	0x80B0	Decrementado a cada ciclo por 1 enquanto diferente de 0

Tabela 6 - Registradores do periférico Timer

Recomenda-se o uso deste periférico através do *syscall* SetTimer, exemplificado na seção 4.

Porta de E/S: A porta de E/S disponibiliza 16 bits, individualmente configuráveis para Entrada ou Saída, para comunicação exterior com o microcontrolador. A porta de E/S consiste, em essência, de 4 registradores: *portData*, que contém, o último valor escrito em sí pelo processador, em seu bits configurados como saída, ou o dado vindo do exterior, se configurado como entrada e habilitados; *portConfig*, que configura os bits correspondentes de *portData* como entrada, para seus bits ativos (1) ou saída, para seus bits inativos (0); *portEnable*, habilita os bits correspondentes de *portData*, seus bits iguais a 1, e desabilita os bits iguais a 0; *portIRQ*, que ativa a geração de interrupções conforme os bits correspondentes de *portData* (Esse registrador só é relevante ao bits 15 a 12 de *portData*, que são conectados às entradas 7 a 4 do PIC, respectivamente. Como não estão conectados ao PIC, os demais bits de *portData* são incapazes de gerar interrupções, independente ao valor do bit correspondente em *portIRQ*). Além dos registradores acima, a porta conta com 16 *tristates*, vinculados a cada bit do registrador *portData* e sua respectiva saída do periférico (interface externa, seta cinza na Figura 1), que são habilitados (individualmente) quando seu bit correspondente em portEnable é igual a 1 (bit habilitado) e seu bit correspondente em *portConfig* é igual a 0 (saída). Adicionalmente, existem mais dois registradores não (diretamente) acessíveis pelo processador: *regSync1* e *regSync2*, a saída deste conectada à entrada daquele, e a saída daquele conectada à entrada de *portData*, a fim de evitar a entrada de dados metaestáveis no registrador *portData* (entre cada bit de *regSync2* e *portData* há também um multiplexador, que seleciona os dados de regSync2 somente se o bit desejado de *portData* estiver habilitado e configurado como entrada, de acordo com registradores *regEnable* e *regConfig*, caso contrário, seleciona como entrada de *portData* o bit correspondente da interface com o processador). A Porta de E/S pode também ser interpretada como 16 periféricos independentes, tendo em comum somente seus bits correspondentes nos registradores descritos acima, visto que podem ser controlados e habilitados de forma individual.

Registrador	Endereço	Funcionalidade (bit a bit)
portData	0x8000	Dado (1 ou 0)
portConfig	0x8001	Entrada (1) / Saída (0)
portEnable	0x8002	Habilita bit (1) / Desabilita bit (0)
portIRQ	0x8003	Habilita interrupção (1) / Desabilita interrupção (0)

Tabela 7 - Registradores do periférico Porta de E/S

```
;   r0 <= 0
    xor r0, r0, r0

;   r1 <= 0xFF00 = 1111111100000000
    ldh r1, #FFh
    ldl r1, #0

;   r2 <= 0xAAAA = 1010101010101010
    ldh r2, #AAh
    ldl r2, #AAh

;   r3 <= 0x5555 = 0101010101010101
    ldh r3, #55h
    ldl r3, #55h

;   r4 <= 0xC000 = 1100000000000000
    ldh r4, #C0h
    ldl r4, #00h

;   r5 <= 0x8003 = &portIRQ
    ldh r5, #80h
    ldl r5, #03h

;   portIRQ <= 0xC000 (Habilta geração de interrupção por bits 15 e 14)
    st r4, r0, r5

;   r5 <= 0x8002 = &portEnable
    subi r5, #1

;   portEnable <= 0xFF00 (Habilita bits mais significativos da Porta de E/S)
    st r1, r0, r5

;   r5 <= 0x8001 = &portConfig
    subi r5, #1

;   portConfig <= 0xAAAA (Seta bits impares como entrada e bits pares como saida)
    st r2, r0, r5

;   r5 <= 0x8000 = &portData
    subi r5, #1

;   portData <= 0x5555 (Escreve 1 nos bits pares, setados como saida) (Escrita so é efetivada nos bits mais significativos)
    st r3, r0, r5
```

Figura 8 - Uso do periférico Porta de E/S

Transmissor Serial UART (UART TX): Transmite serialmente, com um bit de Start = 0, um bit de Parada = 0, sem bit de paridade, o dado, de 8 bits, contido no registrador *TX_DATA*, com uma velocidade de acordo com o registrador *RATE_FREQ_BAUD*, e quando termina a transmissão, escreve 1 no registrador *READY*, sinalizando para o processador que o periférico está apto para realizar outra transmissão. Complementarmente, escreve 0 no registrador *READY* ao começo de uma transmissão, sinalizando ao processador que o periférico está ocupado. O registrador *RATE_FREQ_BAUD* deve conter a taxa entre o clock do microcontrolador e a taxa de símbolos por segundo (em Baud) almejada. O conteúdo de *RATE_FREQ_BAUD* representa a quantidade de tempo em ciclos de clock em que um bit deve ser transmitido, tanto bits de dados como de Start ou Parada.

Registrador	Endereço	Funcionalidade
TX_DATA	0x8080	Dado a ser transmitido (8 bits)
RATE_FREQ_BAUD	0x8081	Taxa entre clock (Hz) e símbolos por segundo (Baud)
READY	0x8082	Sinaliza disponibilidade do periférico

Tabela 8 - Registradores do periférico UART TX

```
;   r0 <= 0
    xor r0, r0, r0

;   Seta RATE_FREQ_BAUD = 869 (0x364) (57600 baud @ 50 MHz)
    ldh r1, #80h
    ldl r1, #81h
    ldh r5, #03h
    ldl r5, #64h
    st r5, r0, r1

;   r1 <= &Ready
    ldh r1, #80h
    ldl r1, #82h

  ReadyLoop: ; Espera transmissor estar disponível

;   r5 <= Ready
    ld r5, r0, r1
    add r5, r0, r5 ; Gera flag de resultado nulo
    jmpzd #ReadyLoop

;   r1 <= &TX_DATA
    ldh r1, #80h
    ldl r1, #80h

;   r5 <= 48 (ASCII '0')
    ldh r5, #0
    ldh r5, #48

;   Transmite caracter '0' (TX_DATA <= '0')
    st r5, r0, r1
```

Figura 9 - Uso do periférico UART TX

Receptor Serial UART (UART RX): Recebe serialmente, segundo o mesmo protocolo de UART TX, um dado de 8 bits, e, a cada novo dado recebido, interrompe o processador (Entrada 1 do PIC). Possui dois registradores: *RX_DATA*, contendo o dado de 8 bits recebido, e *RATE_FREQ_BAUD*, contendo a taxa entre o clock do microcontrolador e a taxa de símbolos (em Baud).

Registrador	Endereço	Funcionalidade
RX_DATA	0x80B0	Dado recebido (8 bits)
RATE_FREQ_BAUD	0x80B1	Taxa entre clock (Hz) e símbolos por segundo (Baud)

Tabela 9 - Registradores do periférico UART RX

```
;   r0 <= 0
    xor r0, r0, r0

;   Seta RATE_FREQ_BAUD = 869 (0x364) (57600 baud @ 50 MHz)
    ldh r1, #80h
    ldl r1, #81h
    ldh r5, #03h
    ldl r5, #64h
    st r5, r0, r1

;   r2 <= 48 (ASCII '0')
    ldh r2, #0
    ldl r2, #48

  WaitFor0Loop: ; Espera dado recebido ser '0'

;   r5 <= Ready
    ld r5, r0, r1

;   Se r5 - 48 = 0, ultimo caracter foi '0'
    sub r5, r5, r2
    jmpzd #WaitFor0LoopBreak
    jmpd #WaitFor0Loop

  WaitFor0LoopBreak:

;   Ultimo dado recebido foi '0'
    halt
```

Figura 10 - Uso do periférico UART RX

4. SYSCALLS

Ao final da execução de uma instrução *syscall*, uma *trap* de código 8 (conforme a Tabela 3) é gerada, o que leva o processador a buscar instruções a partir do endereço de memória contido em *regTSRA*, identificando a trap a ser tratada (código 8), e posteriormente, o *syscall* a ser executado (identificado pelo valor de r1). Feito isso, é chamada a subrotina relativa ao identificador contido em r1 (*syscall* desejado), e a *trap* se dá por tratada, após o término da execução da subrotina chamada, retomando o fluxo de execução normal, com o processador buscando a instrução imediatamente após a instrução de *syscall* executada.

A tabela abaixo contém os *syscalls* do *kernel* desenvolvido, o valor identificador necessário em r1, seus argumentos esperados, assim como uma breve descrição de sua funcionalidade (um exemplo e descrição mais detalhada de cada *syscall* está disponível após a tabela):

Syscall	Numero identificador (em r1)	Argumentos (=) / Retorno (:)
PrintString: Transmite por UART TX uma string terminada com '\0'	0	r2 = Ponteiro para string a ser transmitida
IntegerToString: Converte um inteiro dado em uma string de caracteres ASCII (Em notação decimal)	1	r2 = Inteiro a ser convertido r14 : String equivalente
IntegerToHexString: Converte um inteiro dado em uma string de caracteres ASCII (Em notação hexadecimal)	2	r2 = Inteiro a ser convertido r14 : String equivalente
Delayms: Gasta um valor determinado (em milisegundos) em tempo de processador (Considera um clock de 50 MHz)	3	r2 = Tempo a ser gasto, em milisegundos
IntegerToSSD: Retorna em r14 um dado inteiro (> 0, < 0xF) codificado em notação de display de sete segmentos, considerando os segmentos sendo ativos em 0	4	r2 = Inteiro a ser codificado r14 : Inteiro codificado
Read: Retorna em r14: 0, se o buffer preenchido com caracteres recebidos através do receptor UART ainda não pode ser preenchido (ainda não foi	5	r2 = Ponteiro para string de caracteres do buffer devem ser transferidos r3 = Quantidade de caracteres a serem transferidos do buffer para a string apontada por r2

recebido um caractere ‘/n’); r3, se foi recebido um caractere ‘\n’		r14: Quantidade de caracteres transferidos para string em r2
StringToInteger: Converte uma string de caracteres ASCII em um número inteiro, retornado em r14	6	r2 = Ponteiro para string a ser convertida r14 : Inteiro convertido a partir da string apontada por r2
SetTimer: Programa uma interrupção a ser gerada a uma diferença de tempo (em microsegundos), dada em r2, se essa interrupção deve ser periódica, se r3 != 0, para a função *callback* a ser executada uma vez passada essa diferença de tempo, em r4, e se deve ser executada tal função *callback*, se r5 != 0 (Considera um clock de 50 MHz)	7	r2 = Periodo do timer, em µs r3 = Flag de periodicidade r4 = Ponteiro para função *callback* r5 = Flag de habilitação de callback
WaitForTimer: Retorna 0 enquanto valor de tempo do timer não for esgotado	8	r14: 0, se periodo foi esgotado, caso contrario, 1

Tabela 10 - Syscalls *do* kernel *desenvolvido*

PrintString: Espera, no registrador r2, um ponteiro para uma string, terminada por um caractere ‘\0’, a ser transmitida, um caractere por vez, através do transmissor serial UART.

```
;   r1 <= 0 (Identificador do syscall PrintString)
    ldh r1, #0
    ldl r1, #0

;   r2 <= Ponteiro para string a ser enviada por UART TX
    ldh r2, #StringASerTransmitida
    ldl r2, #StringASerTransmitida

;   Executa syscall ID 0 (PrintString), em r1, e ponteiro para string a ser transmitida, em r2
    syscall
```

Figura 11 - Uso do syscall PrintString *(Omitida TSR e declaração de variáveis)*

A subrotina *PrintString*, chamada dentro da TSR, espera até UART TX estar disponível (sinalizado pelo valor do registrador *Ready*, conforme Tabela 1, acessível no endereço 0x8082), e quando estiver, lê um caractere a partir do endereço de memória dado em r2, armazena-o no registrador TX_DATA de UART TX (endereço 0x8080), e repete esse processo, até ser lido um caractere ‘\0’.

IntegerToString: Converte um valor inteiro, dado em r2, para uma string, retornada em r14, contendo o valor contido em r2 representado em caracteres ASCII. Por exemplo, se dado como argumento o valor 890, é retornado um ponteiro para uma string contendo, sequencialmente, os caracteres: ‘8’, ‘9’, ‘0’ e ‘\0’.

```
;   r1 <= 1 (Identificador do syscall IntegerToString)
    ldh r1, #0
    ldl r1, #1

;   r2 <= Numero a ser convertido, no caso, 890
    ldh r2, #03h
    ldl r2, #7ah

;   Executa syscall ID 1 (IntegerToString), em r1, e valor inteiro a ser convertido, em r2
    syscall

;   Nesse ponto, encontra-se em r14 um ponteiro para uma string contendo os caracteres: ( '8', '9', '0' e '\0')
    halt
```

Figura 12 - Uso do syscall IntegerToString *(Omitida TSR)*

IntegerToHexString: De forma semelhante a IntegerToString, O syscall IntegerToHexString converte um valor inteiro, dado em r2, para uma string de caracteres ASCII, porém, um base hexadecimal. Por exemplo, dado como argumento o valor 255, é retornado em r14 um ponteiro para uma string contendo, sequencialmente, os caracteres: ‘F’, ‘F’, e ‘\0’.

```
;   r1 <= 2 (Identificador do syscall IntegerToHexString)
    ldh r1, #0
    ldl r1, #2

;   r2 <= Numero a ser convertido, no caso, 255
    ldh r2, #0
    ldl r2, #255

;   Executa syscall ID 2 (IntegerToHexString), em r1, e valor inteiro a ser convertido, em r2
    syscall

;   Nesse ponto, encontra-se em r14 um ponteiro para uma string contendo os caracteres: ( 'F', 'F', e '\0')
    halt
```

Figura 13 - Uso do syscall IntegerToHexString *(Omitida TSR)*

Delayms: Gasta um certo tempo, em milisegundos, dado em r2, em tempo de processador.

```
;   r1 <= 3 (Identificador do syscall Delayms)
    ldh r1, #0
    ldl r1, #3

;   r2 <= Numero de milisegundos a serem dispediçados em tempo de processador, nesse exemplo, 10
    ldh r2, #0
    ldl r2, #10

;   Executa syscall ID 3 (Delayms), em r1, e valor de tempo, em milisegundos, em r2
    syscall

;   Esta instrução só será executada 10 ms após a instrução de syscall anterior
    halt
```

Figura 14 - Uso do syscall Delayms *(Omitida TSR)*

IntegerToSSD: Condiciona um valor inteiro, entre 0 e 15, a ser exibido em um display de sete segmentos. Considera que os segmentos são ativos (acesos) quando sua saída digital correspondente é igual a zero. Também considera que o segmento relativo ao ponto (de casa decimal) é correspondente ao bit menos significativo e deve estar sempre desabilitado, independente ao valor passado como argumento. O valor de retorno é dado em r14, devendo ser considerado somente os 8 bits menos significativos. Desses, o bit mais significativo (bit 7) é o correspondente ao segmento A, o segundo mais significativo (bit 6) é o correspondente ao segmento B, e assim sucessivamente. Por exemplo, uma entrada de 0 retorna 00000011, ou seja, todos os segmentos ativos, exceto o segmento G e a casa decimal, formando o desenho do algarismo 0 quando aplicado a um display de sete segmentos.

```
;   r1 <= 4 (Identificador do syscall IntegerToSSD)
    ldh r1, #0
    ldl r1, #4

;   r2 <= Numero a ser codificado, neste exemplo, 8
    ldh r2, #0
    ldl r2, #8

;   Executa syscall ID 4 (IntegerToSSD), em r1, e valor inteiro a ser codificado, em r2
    syscall

;   Neste ponto r14 contem o numero 8 codificado (r14 = xxxxxxxx00000001)
    halt
```

Figura 15 - Uso do syscall IntegerToSSD *(Omitida TSR)*

Read: Enquanto não for recebido um caractere ‘\n’ (Enter) em UART RX, retorna 0, em r14 (Buffer de entrada não está pronto para ser tratado). Quando for recebido um caracter ‘\n’, é inserido em seu lugar um caractere ‘\0’, terminando a string, tornando o *syscall* Read apto a transferir o conteúdo do buffer de entrada para um endereço de memória esperado em r2, em até uma certa quantidade de caracteres, esperada em r3. Feita essa transferência, a posição de começo do buffer (localização no buffer de entrada onde a próxima chamada de Read começará a transferir caracteres) é redefinida para uma posição após o terminador ‘\0’, e então retorna em r14 a quantidade de caracteres transferidos (r3). Recomenda-se que esse *syscall* deve seja utilizado dentro de um loop onde é verificado se o valor de retorno é diferente de 0, se a comparação for verdadeira, o loop é quebrado, e as instruções subsequentes podem ser executadas. Deve-se observar que o *syscall* usado dentro do loop descrito tem comportamento similar à função *scanf* da biblioteca *stdio* da linguagem C, interrompendo a execução do programa até algum valor ser recebido no receptor UART.

```
;   r1 <= 5 (Identificador do syscall Read)
    ldh r1, #0
    ldl r1, #5

;   r2 <= Ponteiro para string onde dados recebidos deve ser salvos
    ldh r2, #stringRecebeDados
    ldl r2, #stringRecebeDados

;   r3 <= Quantidade de caracteres a serem transferidos do buffer de entrada para "stringRecebeDados"
    ldh r3, #0
    ldl r3, #3

  ReadLoop:

;   Executa instrução syscall enquanto dados não forem transferidos para "stringRecebeDados"
    syscall
    add r14, r0, r14 ; Gera flag de resultado nulo
    jmpzd #ReadLoop

;    Somente executa esta instrução quando o valor de retorno do syscall Read for diferente de 0
;   ou seja, enquanto 3 caracteres não forem transferidos do buffer de entrada para "stringRecebeDados"
    halt
```

Figura 16 - Uso do syscall Read *(Omitida TSR e declaração de variáveis)*

StringToInteger: Ao contrário do syscall IntegerToString, StringToInteger converte uma string de caracteres ASCII em um valor inteiro. O ponteiro para a string de entrada deve ser dado em r2, e o valor inteiro convertido é retornado em r14.

```
;   r1 <= 6 (Identificador do syscall StringToInteger)
    ldh r1, #0
    ldl r1, #6

;   r2 <= Ponteiro para string onde numero a ser convertido está localizado
    ldh r2, #stringNumeroEmASCII
    ldl r2, #stringNumeroEmASCII

;   Executa syscall ID 6 (StringToInteger), em r1, e ponteiro para string com valor a ser convertido, em r2
    syscall

;   Supondo que "stringNumeroEmASCII" aponte para ('5', '2', '0', '\0'), após o syscall r14 conterá o valor 520
    halt
```

Figura 17 - Uso do syscall StringToInteger *(Omitida TSR e declaração de variáveis)*

SetTimer: Programa o periférico Timer para gerar uma interrupção de alta prioridade a uma certa diferença de tempo, dada em r2, em microsegundos. Também pode ser informado se essa interrupção deve ser periódica, se r3 = 1, o endereço de memória de uma subrotina a ser executada quando passada esta diferença de tempo (*callback*), em r4, e se tal subrotina deve ser executada, se r5 = 1. Deve-se observar que a lógica de cálculo de passagem de tempo considera um sinal de clock de 50 MHz.

```
;   r1 <= 7 (Identificador do syscall SetTimer)
    ldh r1, #00h
    ldl r1, #07h

;   r2 <= Periodo em microsegundos
    ldh r2, #0
    ldl r2, #200

;   r3 <= Flag de periodicidade do timer
    ldh r3, #0
    ldl r3, #1

;   r4 <= Ponteiro para função de callback
    ldh r4, #TimerCallback
    ldl r4, #TimerCallback

;   r5 <= Flag de callback
    ldh r5, #0
    ldl r5, #1

;   A cada 200 us será lançada uma nova interrupção e executada a subrotina "TimerCallback"
    syscall
```

Figura 18 - Uso do syscall SetTimer *(Omitida TSR e ISR)*

WaitForTimer: A ser usado juntamente ao *syscall* SetTimer, funciona de modo similar ao *syscall* Read, retornando 0 em r14 enquanto o valor de tempo sendo contabilizado no periférico Timer não estiver passado. Pode ser interpretado como uma forma de usar o periférico Timer por *polling*, ao invés da utilização de uma função *callback*.

```
;   r1 <= 7 (Identificador do syscall SetTimer)
    ldh r1, #00h
    ldl r1, #07h

;   r2 <= Periodo em microsegundos
    ldh r2, #0
    ldl r2, #200

;   r3 <= Flag de periodicidade do timer
    ldh r3, #0
    ldl r3, #0

;   r5 <= Flag de callback
    ldh r5, #0
    ldl r5, #0

;   Em uma diferença de tempo de 200 us será lançada uma interrupção (não periódica)
    syscall

;   r1 <= 8 (Identificador do syscall WaitForTimer)
    ldh r1, #0
    ldh r1, #8

  WaitForTimerLoop:

;   Executa syscall ID 8 (WaitForTimer)
    syscall
    add r14, r0, r14 ; Gera flag de resultado nulo
    jmpzd #WaitForTimerLoop

;   Somente executa esta instrução se passados 200 us desde o primeiro syscall (SetTimer)
    halt
```

Figura 19 - Uso do syscall WaitForTimer *(Omitida TSR e ISR)*

5. ESTRUTURA DOS ARQUIVOS FONTE

Junto a esse arquivo de documentação, estão disponibilizados os códigos fonte desenvolvidos. Os arquivos estão estruturados dentro das pastas:

Documentação: Nesta pasta devem estar contidos este documento, a documentação do processador R8 original, assim como os exemplos mostrados ao longo deste arquivo, dentro do subdiretório Exemplos. Também está incluída uma versão em maior resolução da Figura 2.

Montador: Nesta estão os arquivos necessários para a geração de imagem de memória: R8_sim.jar e R8.arq. Os procedimentos para execução de um novo código, como alguma modificação no programa exemplo disponibilizado, podem ser deduzidos a partir da seção 6 deste documento.

Programa Exemplo: Aqui estão, o código fonte do programa exemplo (ProgramaExemplo.asm), as imagens de memória geradas pelo montador, coberto no item acima, (ProgramaExemplo.bin, a ser transmitida por UART e programada a partir de um *bitstream* do microcontrolador já existente no FPGA, e ProgramaExemplo_BRAM.txt, se desejado ter a imagem do programa gravada na RAM em tempo de síntese, mais detalhes no item Sources), e um arquivo de texto auxiliar, de onde é possível saber aonde na memória se encontra cada instrução e os dados do programa (ProgramaExemplo.txt).

Sources: A pasta principal do projeto, nela estão contidos os arquivos VHDL para a síntese do microcontrolador. Aqui estão as descrições de cada periférico (UART_RX.vhd, UART_TX.vhd, PortaBidirecional.vhd, InterruptController.vhd, PriorityEncoder.vhd (instanciado dentro do PIC) e Timer.vhd), do processador R8 modificado (R8.vhd), da memória (Memory.vhd, relativo tanto à memória RAM quanto memória ROM), do microcontrolador em si, (R8_uC.vhd, onde são instanciados os periféricos, processador, memórias RAM e ROM, e feitas as interconexões entre tais, conforme a Figura 2, assim como especificados os endereços de 4 bits dos registradores individuais dentro de cada periférico), e finalmente, da entidade de mais alto nível R8_uC_TOPLVL.vhd, onde é instanciado o microcontrolador, e conectados a tal o *Digital Clock Manager*, (ClockManager.vhd, necessária biblioteca *unisim* para simulação, disponibilizada como unisim.zip), a fim de gerar um sinal de clock de 50 MHz a partir do clock de 100 MHz provido pela placa, e Sincronizador de Reset (ResetSynchronizer.vhd). A geração do *bitstream* é feita a partir do arquivo R8_uC_TOPLVL.vhd, que infere o DCM como componente do FPGA Spartan6 e gera a lógica necessária conforme os arquivos fonte. No arquivo R8_uC_TOPLVL.vhd é possível também informar as imagens desejadas para as memórias, através dos *generics* ROM_IMAGE e RAM_IMAGE e especificar os endereços de 4 bits desejados dos diferentes periféricos, também através de *generics*. Estão também disponíveis aqui os arquivos com os códigos fonte do *kernel* e do programa de programação da memória RAM contido na memória ROM.

Sintese: Aqui estão contidos os *reports* de área e frequência, produzidos a partir do componente XST do software Xilinx ISE.

6. EXECUÇÃO DO PROGRAMA EXEMPLO NA PLACA NEXYS 3

Junto a este documento é disponibilizado um programa exemplo que utiliza todas as funcionalidades do projeto de microcontrolador desenvolvido. O programa executa o algoritmo *Bubble Sort* em um *array*, de acordo com as seguinte informações requisitadas ao usuário, recebidas através de comunicação serial UART: Tamanho do *array*, Elementos do *array*, e ordenação do *array* (0 para crescente, 1 para decrescente). Além disso é exibido um contador com período de 1 segundo nos *displays* mais a direita da placa, assim como um contador manual, controlado pelos botões *UP* e *DOWN* da placa, exibido nos *displays* mais a esquerda.

1. **Programar o FPGA via o *software* Digilent Adept com o arquivo ROM.bit**:

Este arquivo consiste de um *bitstream* com o microcontrolador desenvolvido, contido em em sua memória ROM um pequeno programa que possibilita o carregamento de dados na memória RAM (no caso, o programa exemplo especificado acima) a partir do periférico UART RX.

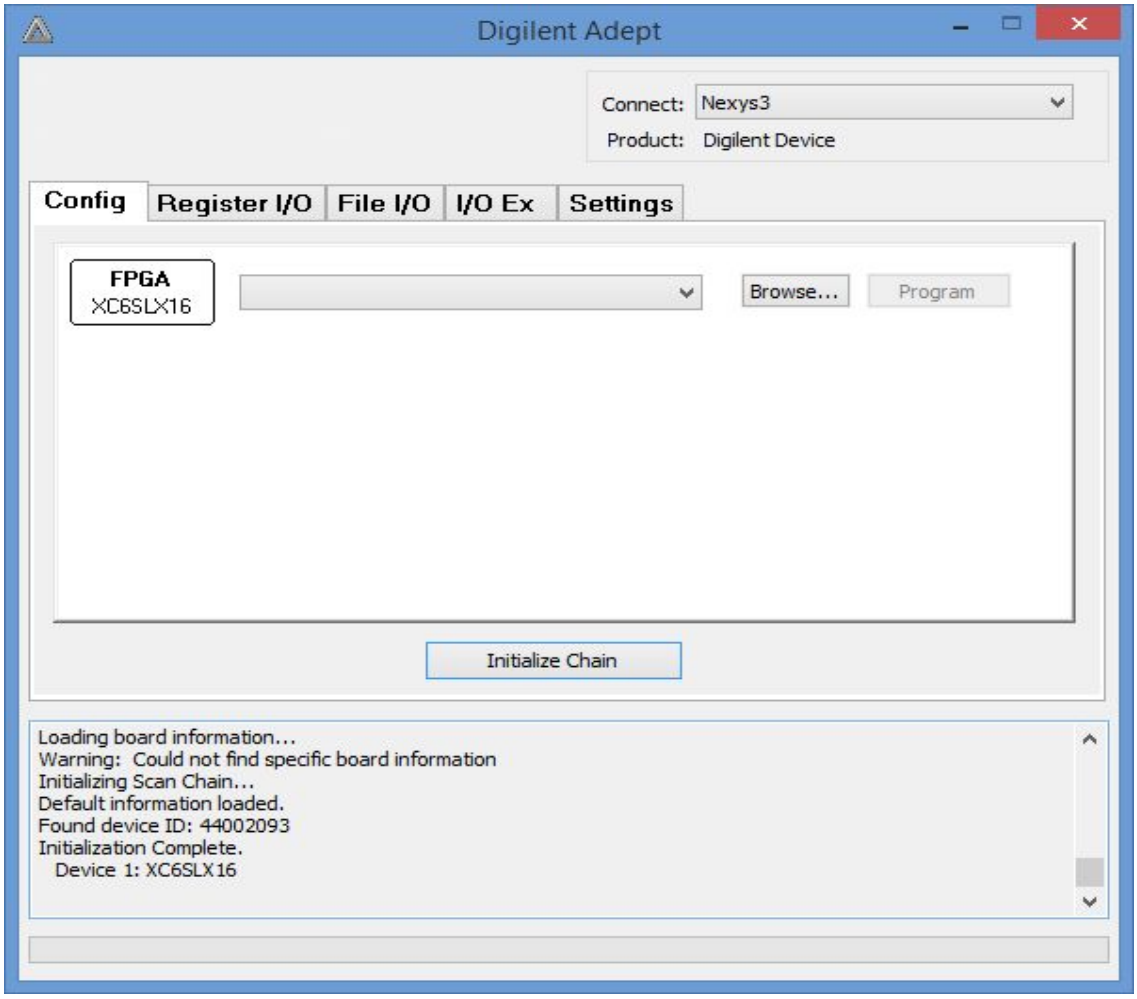

Figura 20 - Software ADEPT com placa Nexys 3 conectada ao computador via conector USB PROG

Após ligar a placa, e conectar o USB PROG da placa ao computador, no canto superior direito deve estar sendo exibida o nome da placa Nexys 3 conectada, e no menu *dropdown* central, deve então ser selecionado o arquivo ROM.bit, e pressionado o botão “Program”. Após um certo tempo, deve ser exibido no console do *software* uma mensagem de sucesso, informando que o FPGA foi programado com sucesso.

O *software* Adept está disponível em:
https://store.digilentinc.com/digilent-adept-2-download-only/

2. **Configurar o terminal serial:**

Deve ser utilizado um *software* de terminal serial, a fim de enviar ao microcontrolador o arquivo binário contendo o programa exemplo (ou qualquer programa a ser executado), assim como realizar a Entrada/Saída de dados necessária. A comunicação serial deve ser de acordo com o protocolo do Receptor (UART RX) e Transmissor (UART TX) implementados: bit de Start = 0, bit de Parada = 1, sem bit de paridade, e 8 bits de dados por transmissão. No programa exemplo é usada uma taxa de símbolos de 57600 Baud, que deve também ser a taxa utilizada pelo terminal serial. Essa taxa pode ser modificada no programa exemplo (através do registrador *RATE_FREQ_BAUD*, conforme descrito na seção 3), desde que seja utilizada a mesma taxa de símbolos no software de terminal serial. Neste tutorial, será usado o *software* HyperTerminal.

Para o uso do terminal serial, é preciso instalar um driver no Windows, que emula uma porta COM a partir de uma porta USB do computador. Tal porta USB deve ser ligada por cabo ao USB UART da placa. Se a placa estiver sendo alimentada pelo conector USB de programação, o cabo conectado ao conector USB PROG não deve ser desplugado, caso contrário, basta a conexão por USB UART ao computador.

Dentro do *HyperTerminal*, sob o meu File, clique em New Connection. Após isso, deverá ser aberta uma janela intitulada Connect To. Dentro desta, deve ser selecionada a porta COM relativa ao driver instalado, e então configurados os parâmetros de comunicação, a partir da janela aberta ao clicar no botão “Configure…”. A taxa de bits deverá ser de 57600 Baud, com 8 bits de dados, sem bit de paridade, 1 bit de parada, sem controle de fluxo. Informados tais dados, clique OK nas duas janelas. Feito isto, o terminal serial estará configurado.

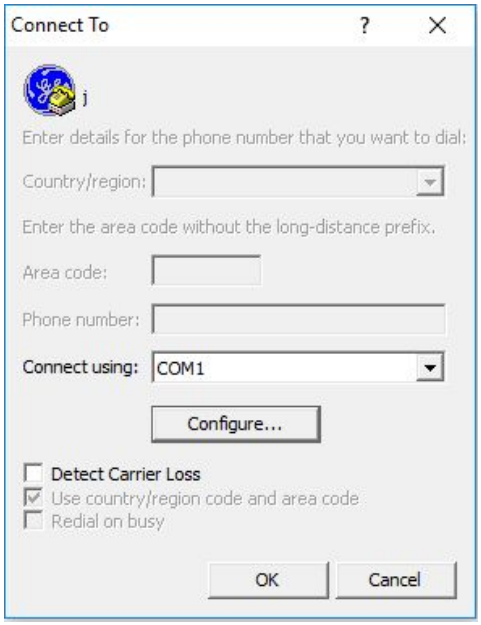

Figura 21 - Janela "Connect To"

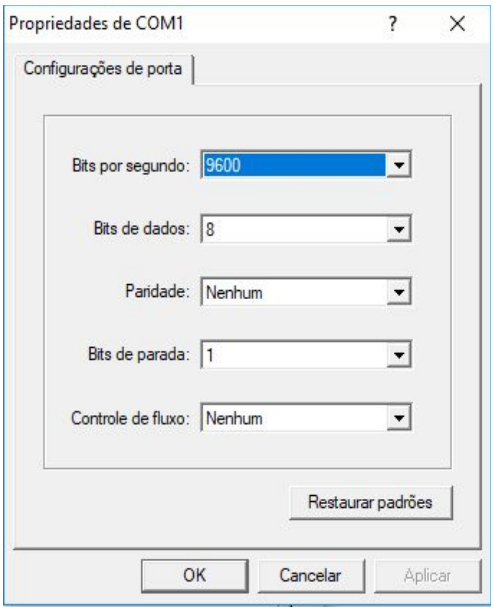

Figura 22 - Parâmetros de Comunicação

O driver para emulação da porta COM, necessário para a comunicação via UART está disponível em:
https://www.ftdichip.com/Drivers/VCP.htm

O software HyperTerminal está disponível em:
https://www.hilgraeve.com/hyperterminal/

3. **Gerar imagem da memória com o software R8_sim.jar**:

Com o arquivo R8.arq, que contém as especificações das instruções a serem montadas pelo montador a partir do código fonte, na mesma pasta do arquivo R8_sim.jar, deve-se executar o arquivo R8_sim.jar, e dentro do mesmo, no menu File, selecionar o item Load, e na janela aberta, selecionar o arquivo ProgramaExemplo.asm (código fonte do programa exemplo). Após um tempo, deve ser gerada uma mensagem de erro, irrelevante para o propósito deste tutorial, e na pasta que contém o arquivo ProgramaExemplo.asm, alguns novos arquivos, dentre eles, ProgramaExemplo.bin, que é a imagem da memória ser enviada pelo terminal serial com o programa a ser executado, devem estar presentes.

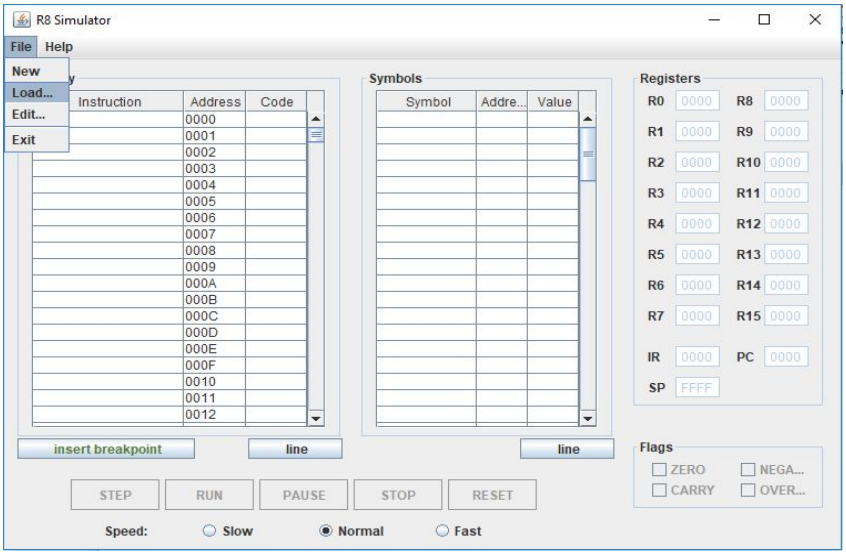

Figura 23 - Montador

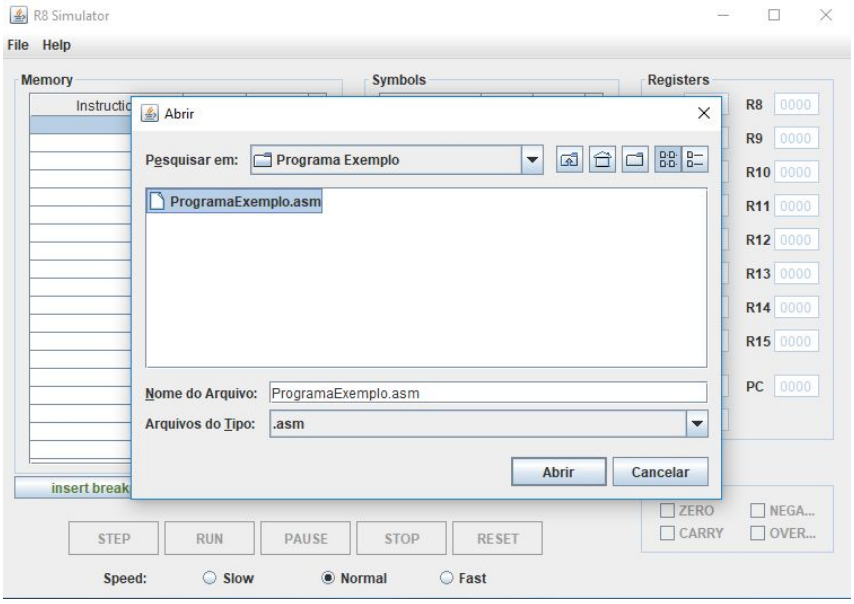

Figura 24 - Arquivo fonte do programa exemplo

Para a execução do software R8_sim.jar é necessário Java, disponível em: https://www.java.com/en/download/

4. **Ativar o modo de programação do microcontrolador**:

Para ativar o modo de programação do microcontrolador, basta colocar o *switch* mais a direita (mais próximo aos botões e displays) em sua posição ativa (cima).

5. **Enviar a imagem de memória gerada através do terminal serial**:

No *software* HyperTerminal, sob o menu Transfer, selecione o item Send Text File, e na janela aberta, o arquivo ProgramaExemplo.bin gerado no passo 2. O LED vinculado ao receptor serial da placa deve estar piscando, sinalizando a transferência de dados entre o computador e o microcontrolador. Quando este parar de piscar, a transferência da imagem da memória terminou, após isso, é possível prosseguir ao próximo passo.

6. **Desativar o modo de programação do microcontrolador:**

Basta retornar o switch mais a direita a sua posição não-ativa (baixo).

7. **Utilizar o software normalmente**:

Feito isso, o microcontrolador estará em modo de execução, executando o programa exemplo, transferido via UART. É possível inserir os valores necessários para o *Bubble Sort* via terminal serial e pressionar os botões da placa para manipular o contador manual.