Trabalho Final

Arquitetura de Computadores

Prof. Pedro Botelho

2º Semestre de 2025

1 Descrição

Seu trabalho será implementar um programa (em linguagem C, C++, Java ou Go) que simule a execução de um **processador RISC**, contendo as instruções mostradas na tabela fornecida na Seção 3.

Considere que o processador possui uma palavra de dados e de instrução de 16 bits, e 16 registradores de trabalho (R0-R15), onde R14 é usado como **ponteiro de pilha** (SP) e R15 é usado como **contador de programa** (PC). Adicionalmente, o processador possui o **registrador de instrução** (IR) e **registrador de estado** (FLAGS).

2 Especificação do Projeto

O simulador deverá receber como entrada um arquivo de texto contendo o código a ser executado, em notação hexadecimal (ou Assembly, vide Seção 4). Este arquivo representa as posições de memória a serem preenchidas, onde cada linha corresponde a um endereço e seu conteúdo, no formato <endereço> <conteúdo>. O preenchimento da memória de dados é **opcional**, podendo ser no mesmo arquivo, após as instruções.

Abaixo temos um exemplo da entrada do simulador, onde cada linha tem o endereço (em hexadecimal) e o hexadecimal referente à instrução (ou ao dado, no endereço 000B). Todo o restante pode ser ignorado (o Assembly equivalente e comentários):

```
0000 00B4 MOV R0, #11          // R0 = 0xB
0001 A002 LDR R10, [R0, #0]    // R10 = 0xF000
0002 1A22 LDR R1, [R10, #2]    // R1 <= IN1
0003 2A22 LDR R2, [R10, #2]    // R2 <= IN2
0004 3125 ADD R3, R1, R2       // R3 = IN1 + IN2
0005 3A33 STR R3, [R10, #3]    // OUT <= R3
0006 030D CMP R3, R0           // R3 < 11?
0007 BFA1 JLT -6               // Yes? Repeat!
0008 9414 MOV R9, 0x41         // No? R9 = 'A'
0009 1A93 STR R9, [R10, #1]    // Print 'A'
000A FFFF HALT                 // Stop!
000B F000 IO_BASE              // Points to I/O base address
```

2.1 Estrutura do Processador

O processador a ser simulado usa o modelo de **von Neumann** (dados e instruções na mesma memória). Ainda, cada posição de memória guarda 16 bits. A memória principal está mapeada para os endereços entre 0x0000 a 0x2000 (8192 posições de 2 bytes cada, totalizando 16KB), e as portas de E/S estão mapeadas a partir do endereço 0xF000 (vide seção 2.3).

O processador possui apenas duas *flags*:

- Flag **zero**: Ativada quando o resultado da ULA é zero.
- Flag **carry**: Ativada quando o resultado da ULA tem um vai-um do bit 15 para o 16 ativo

As *flags* só são modificadas por instruções da ULA. De forma a simplificar, a ***flag* carry** pode ser ativa quando **Rd for menor que zero** (ou se Rn for maior que Rm, para a instrução CMP), o que resulta no mesmo que verificar o bit do vai-um (em comparações sem sinal). Já a ***flag* zero** pode ser ativa quando **Rd for igual a zero** (ou se Rn for igual a Rm, para a instrução CMP). Instruções que não são da ULA não atualizam as *flags*.

Os saltos do processador são baseados em **comparações com sinal** (signed), para simplificar. Portanto, apenas as *flags* zero (Z) e *carry* (C) serão usadas. Ao usar uma instrução de salto condicional, J<cond>, a condição será avaliada com base nas *flags* de estado. Apenas as instruções da ULA modificam as *flags*.

2.2 Estrutura do Simulador

O simulador segue a mesma base daquele implementado no laboratório 3. O programa deve executar até encontrar uma instrução **HALT**. Ao final da execução, o simulador deverá apresentar o conteúdo (em notação hexadecimal) dos seguintes componentes:

- **Registradores**: Todos iniciam zerados (R0-R13, PC e PC).
- **Memória de Dados**: Assuma que a faixa de endereços de 0x0000 a 0x2000 está disponível (endereços fora da faixa são lidos como zero) e que a memória está inicialmente zerada.
  - No mesmo formato do arquivo de entrada (<endereço> <conteúdo>).
  - Devem ser exibidas apenas as posições que forem acessadas pelo código (desconsiderando a pilha e as instruções).
- **Pilha**: Do tipo descendente (de um endereço maior para um menor) e cheia (SP aponta para o último elemento empilhado).

  - No mesmo formato do arquivo de entrada (<endereço> <conteúdo>).
  - Assuma que o ponteiro de pilha com o valor inicial de 0x2000 da memória, e que as posições da pilha possuem também um tamanho de 16 bits.
  - Devem ser exibidas as posições utilizadas apenas se o ponteiro SP não estiver em sua posição original (0x2000).

- **Flags**: Assuma que as todas iniciam zeradas.
  - Exiba os valores finais das *flags carry* e zero.

O simulador também deve compreender ***breakpoints*** (assim como foi feito no laboratório 3). Ou seja, o usuário diz ao simulador (via parâmetros) quais instruções (no endereço especificado) devem ser verificadas. Assim, sempre que o endereço da instrução atual coincidir com o endereço do *breakpoint* o simulador deve exibir todas estas informações. Quando a instrução HALT for executada, essas informações também devem ser mostradas.

2.3 Sistema de Entrada e Saída (E/S)

O simulador deve fornecer ao usuário um sistema básico de entrada e saída mapeado em memória.

- 0xF000 - Entrada **CHAR_IN** (simulador lê um caractere da entrada padrão e.g. assim como na função *scanf*)
- 0xF001 - Saída **CHAR_OUT** (simulador escreve um caractere na saída padrão e.g. assim como na função *printf*)
- 0xF002 - Entrada **INT_IN** (simulador lê um inteiro da entrada padrão e.g. assim como na função *scanf*)
- 0xF003 - Saída **INT_OUT** (simulador escreve um inteiro na saída padrão e.g. assim como na função *printf*)

Segue abaixo um exemplo em código Assembly que acessa o sistema de E/S para ler dois inteiros. Se a soma destes for maior que 10 finaliza o programa. Se não for, reinicia:

```
0000 00A4 MOV R0, #10          // R0 = 0xA
0001 A002 LDR R10, [R0, #0]    // R10 = 0xF000
0002 1A22 LDR R1, [R10, #2]    // R1 <= IN1
0003 2A22 LDR R2, [R10, #2]    // R2 <= IN2
0004 3125 ADD R3, R1, R2       // R3 = IN1 + IN2
0005 3A33 STR R3, [R10, #3]    // OUT <= R3
0006 030D CMP R3, R0           // R3 < 10?
0007 BFA1 JLT -6               // Yes? Repeat!
0008 FFFF HALT                 // No? Stop!
000A F000 IO_BASE
```

3 Conjunto de Instruções

O processador que será simulado possui 20 instruções simples de 16 bits cada. Os 4 bits inferiores (bits 0 a 3) das instruções indicam o código da operação (*opcode*). Ainda, os bits 15-14 das instruções J<cond> indicam a **condição** para o salto, ou seja, o valor das *flags* para que o salto aconteça (instrução JMP sempre salta):

- JEQ (*Jump if equal*): Salta quando a *flag* zero está ativa
- JNE (*Jump if not equal*): Salta quando a *flag* zero não está ativa
- JLT (*Jump if less than*): Salta quando a *flag* zero não está ativa e a *carry* está
- JGE (*Jump if greater or equal*): Salta quando a *flag carry* não está ativa e a zero está

As instruções suportam apenas um modo de endereçamento cada, sendo eles **registradores**, **memória** e **imediato**, conforme se vê na tabela abaixo:

Instruções	**15 14 13 12**	**11 10 9 8**	**7 6 5 4**	**3 2 1 0**
JMP	Imediato			*Opcode*
J<cond>	Cond.	Imediato		*Opcode*
ADDI, SUBI, LDR, SHR, SHL	Rd	Rm	Imediato	*Opcode*
STR	Imediato	Rm	Rn	*Opcode*
MOV	Rd	Imediato		*Opcode*
ADD, SUB, AND, OR	Rd	Rm	Rn	*Opcode*
CMP	-	Rm	Rn	*Opcode*
PUSH	-		Rn	*Opcode*
POP	Rd	-		*Opcode*
HALT	0xFFFF			

O processador possui 16 registradores, logo, 4 bits para endereçá-los, em 3 campos padronizados (Rd, Rm e Rn). Abaixo estão as instruções que o processador deverá suportar:

Instrução	**Descrição**	**Tipo**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
JMP #Im	PC = PC + #Im	JUMP	Im_{11}	Im_{10}	Im_9	Im_8	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	0	0	0
JEQ #Im	If ¬Z, then PC = PC + #Im	JUMP	0	0	Im_9	Im_8	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	0	0	1
JNE #Im	If Z, then PC = PC + #Im	JUMP	0	1	Im_9	Im_8	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	0	0	1
JLT #Im	If ¬Z and C, then PC = PC + #Im	JUMP	1	0	Im_9	Im_8	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	0	0	1
JGE #Im	If Z or ¬C, then PC = PC + #Im	JUMP	1	1	Im_9	Im_8	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	0	0	1
LDR Rd, [Rm, #Im]	Rd = MEM[Rm + #Im]	LOAD	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Im_3	Im_2	Im_1	Im_0	0	0	1	0
STR Rn, [Rm, #Im]	MEM[Rm + #Im] = Rn	STORE	Im_3	Im_2	Im_1	Im_0	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	0	0	1	1
MOV Rd, #Im	Rd = #Im	MOVE	Rd_3	Rd_2	Rd_1	Rd_0	Im_7	Im_6	Im_5	Im_4	Im_3	Im_2	Im_1	Im_0	0	1	0	0
ADD Rd, Rm, Rn	Rd = Rm + Rn	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	0	1	0	1
ADDI Rd, Rm, #Im	Rd = Rm + #Im	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Im_3	Im_2	Im_1	Im_0	0	1	1	0
SUB Rd, Rm, Rn	Rd = Rm - Rn	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	0	1	1	1
SUBI Rd, Rm, #Im	Rd = Rm - #Im	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Im_3	Im_2	Im_1	Im_0	1	0	0	0
AND Rd, Rm, Rn	Rd = Rm AND Rn	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	1	0	0	1
OR Rd, Rm, Rn	Rd = Rm OR Rn	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	1	0	1	0
SHR Rd, Rm, #Im	Rd = Rm >> #Im	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Im_3	Im_2	Im_1	Im_0	1	0	1	1
SHL Rd, Rm, #Im	Rd = Rm << #Im	ALU	Rd_3	Rd_2	Rd_1	Rd_0	Rm_3	Rm_2	Rm_1	Rm_0	Im_3	Im_2	Im_1	Im_0	1	1	0	0
CMP Rm, Rn	Z = (Rm = Rn); C = (Rm < Rn)	ALU	-	-	-	-	Rm_3	Rm_2	Rm_1	Rm_0	Rn_3	Rn_2	Rn_1	Rn_0	1	1	0	1
PUSH Rn	SP- -; MEM[SP] = Rn	STACK	-	-	-	-	-	-	-	-	Rn_3	Rn_2	Rn_1	Rn_0	1	1	1	0
POP Rd	Rd = MEM[SP]; SP++	STACK	Rd_3	Rd_2	Rd_1	Rd_0	-	-	-	-	-	-	-	-	1	1	1	1
HALT	Para a execução	CONTROL	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Apenas 3 instruções utilizam extensão de sinal no imediato (JMP, J<cond> e MOV), enquanto as outras utilizam extensão de zero. Toda vez que uma instruções é buscada seu PC é incrementado em um, devendo sempre apontar para a **próxima instrução** antes de ser executado. Isso deve ser levado em consideração ao computar os saltos.

4 Avaliação e Entrega

O trabalho **deverá** ser feito em equipes de **5 pessoas** (a não ser que só restem 4 ou menos pessoas na turma sem equipe, sendo esta a **única exceção**), e deverá será apresentado ao professor, onde será submetido a vários casos de teste específicos. Serão verificadas as informações mostradas ao final da execução (e sempre que ocorrer um *breakpoint*).

4.1 Entrega do Projeto

O projeto deverá ser **submetido** no Moodle até o dia **16/01**, até o horário de **início da aula**. As apresentações podem ocorrer em qualquer momento, sendo o prazo final o **horário de término**, da aula no dia 16/01. Portanto, se muitas equipes deixarem para apresentar no último momento, quem ficar para apresentar após o término da aula estará automaticamente com **nota zero** na AP3, mesmo tendo enviado o projeto para o Moodle. Portanto, projetos não apresentados valem zero.

As **apresentações** poderão ser feitas em **qualquer dia**, devendo apenas combinar com o professor o dia e horário. Os horários de aula dos dias 15/01 e 16/01 serão direcionados para as apresentações presenciais em sala de aula. Quem desejar apresentar remotamente deve marcar para outro dia.

4.2 Método de Pontuação

Durante a apresentação o professor pode fazer perguntas sobre o projeto a qualquer um dos membros da equipe, podendo deduzir pontos caso a resposta não seja correta. Qualquer ocorrência de plágio acarretará em **nota zero** para todas as equipes involvidas.

O projeto valerá até **12 pontos**, substituindo a nota da AP3. Destes 12 pontos, 10 pontos serão divididos em 5 casos de teste:

- Caso 1 (Instruções básicas): 2 pontos
- Caso 2 (Acesso à memória): 2 pontos
- Caso 3 (Saltos): 2 pontos
- Caso 4 (Operações de E/S): 2 pontos
- Caso 5 (Programa completo): 2 pontos

Os outros 2 pontos serão relativos à **tradução** do código em Assembly, devendo o projeto ser capaz traduzir o Assembly para código de máquina para que a execução seja possível. Esse código Assembly deve ser capaz de entender etiquetas (*labels*) de código (para saltos) e etiquetas de dados.

```
L1: ADD R0, R1, R2  // label de codigo
    BEQ #L1
...
    MOV R0, #L2
L2: 200             // label de dado
```

Caso a equipe deseje escrever só o simulador, a nota máxima que poderá ser alcançada será 10. Sendo assim, o tradutor equivale a 2 pontos extras, e o simulador aos 10 pontos da AP3.

Caso deseje fazer os dois, o projeto deverá conter dois programas separados: um **tradutor** (ou *Assembler*) que obtém o código Assembly e o converte em código de máquina (em hexadecimal) e um **simulador** que obtém o código de máquina e realiza a simulação. A execução então segue o seguinte formato:

```
./assembler code.asm code.hex
./sim code.hex 5
```

4.3 Exemplo de Entrada

Seguem abaixo alguns exemplos de entrada em hexadecimal (e seu Assembly equivalente), com um único *breakpoint* cada, na instrução HALT:

4.3.1 Exemplo 1

Entrada:

```
0000 0054 MOV R0, #5       // R0 = 0x0005
0001 1056 ADDI R1, R0, #5  // R1 = 0x000A
0002 2158 SUBI R2, R1, #5  // R2 = 0x0005
0003 3125 ADD R3, R1, R2   // R3 = 0x000F
0004 4327 SUB R4, R3, R2   // R4 = 0x000A
0005 540A OR R5, R4, R0    // R5 = 0x000F
0006 6509 AND R6, R5, R0   // R6 = 0x0005
0007 752C SHL R7, R5, #2   // R7 = 0x003C
0008 872B SHR R8, R7, #2   // R8 = 0x000F
0009 FFFF HALT             // Stops CPU!
```

Saída:

```
R0 = 0x0005
R1 = 0x000A
R2 = 0x0005
```

```
R3 = 0x000F
R4 = 0x000A
R5 = 0x000F
R6 = 0x0005
R7 = 0x003C
R8 = 0x000F
R9 = 0x0000
R10 = 0x0000
R11 = 0x0000
R12 = 0x0000
R13 = 0x0000
R14 = 0x2000
R15 = 0x000A
Z = 0
C = 0
```

4.3.2 Exemplo 2

Entrada:

```
0000 00C4 MOV R0, #var1          // R0 = 0x000C
0001 1002 LDR R1, [R0, #0]       // R1 = [0x0C] = 0x0A
0002 2012 LDR R2, [R0, #1]       // R2 = [0x0D] = 0x14
0003 3022 LDR R3, [R0, #2]       // R3 = [0x0F] = 0x1E
0004 001E PUSH R1                // [SP - 1] = 0x0A, SP = 0x1FFF
0005 002E PUSH R2                // [SP - 2] = 0x14, SP = 0x1FFE
0006 003E PUSH R3                // [SP - 3] = 0x1E, SP = 0x1FFD
0007 4125 ADD R4, R1, R2         // R4 = 0x1E
0008 4435 ADD R4, R4, R3         // R4 = 0x3C
0009 3043 STR R4, [R0, #3]       // [0x0B] = 0x3C
000A 500F POP R5                 // R5 = 0x1E
000B FFFF HALT                   // Stops CPU!
000C 000A var1: 0x0A             // 10
000D 0014 var2: 0x14             // 20
000E 001E var3: 0x1E             // 30
000F 0000 var4: 0
```

Saída:

```
R0 = 0x000C
R1 = 0x000A
R2 = 0x0014
R3 = 0x001E
R4 = 0x003C
R5 = 0x001E
R6 = 0x0000
R7 = 0x0000
R8 = 0x0000
R9 = 0x0000
R10 = 0x0000
R11 = 0x0000
```

```
R12 = 0x0000
R13 = 0x0000
R14 = 0x1FFE
R15 = 0x000C
Z = 0
C = 0
[0x000C] = 0x000A
[0x000D] = 0x0014
[0x000E] = 0x001E
[0x0010] = 0x001E
[0x1FFF] = 0x000A
[0x1FFE] = 0x0014
```

4.3.3 Exemplo 3

Entrada:

```
0000 00A4 MOV R0, #10          // R0 = 0x000A (addr of size)
0001 3004 MOV R3, #0           // R3 = 0 (sum accumulator)
0002 1002 LDR R1, [R0, #0]     // R1 = [0x0A] = 5 (loop count)
0003 0016 ADDI R0, R0, #1      // R0++ (point to next item)
0004 2002 LDR R2, [R0, #0]     // R2 = [R0] (load value)
0005 3325 ADD R3, R3, R2       // R3 = R3 + R2 (add to sum)
0006 1118 SUBI R1, R1, #1      // R1— (decrement count)
0007 7FB1 JNE #-5              // If R1 != 0, jump to 0x0003
0008 1033 STR R3, [R0, #1]     // [R0+1] = R3 (store result)
0009 FFFF HALT                 // Stops CPU!
000A 0005 size: 5
000B 0001 arr0: 1
000C 0002 arr1: 2
000D 0003 arr2: 3
000E 0004 arr3: 4
000F 0005 arr4: 5
0010 0000 res: 0
```

Saída:

```
R0 = 0x000F
R1 = 0x0000
R2 = 0x0005
R3 = 0x000F
R4 = 0x0000
R5 = 0x0000
R6 = 0x0000
R7 = 0x0000
R8 = 0x0000
R9 = 0x0000
R10 = 0x0000
R11 = 0x0000
R12 = 0x0000
R13 = 0x0000
```

```
R14 = 0x2000
R15 = 0x000A
Z = 1
C = 0
[0x000A] = 0x0005
[0x000B] = 0x0001
[0x000C] = 0x0002
[0x000D] = 0x0003
[0x000E] = 0x0004
[0x000F] = 0x0005
[0x0010] = 0x000F
```

4.3.4 Exemplo 4

Entrada:

```
0000 00B4 MOV R0, #11          // R0 = 0xB
0001 A002 LDR R10, [R0, #0]    // R10 = 0xF000
0002 1A22 LDR R1, [R10, #2]    // R1 <= IN1
0003 2A22 LDR R2, [R10, #2]    // R2 <= IN2
0004 3125 ADD R3, R1, R2       // R3 = IN1 + IN2
0005 3A33 STR R3, [R10, #3]    // OUT <= R3
0006 030D CMP R3, R0           // R3 < 11?
0007 BFA1 JLT #-6               // Yes? Repeat!
0008 9414 MOV R9, 0x41         // No? R9 = 'A'
0009 1A93 STR R9, [R10, #1]    // Print 'A'
000A FFFF HALT                 // Stops CPU!
000B F000 IO_BASE
```

Saída:

```
IN => 1
IN => 1
OUT <= 2
IN => 6
IN => 6
OUT <= 12
OUT <= A
R0 = 0x000B
R1 = 0x0006
R2 = 0x0006
R3 = 0x000C
R4 = 0x0000
R5 = 0x0000
R6 = 0x0000
R7 = 0x0000
R8 = 0x0000
R9 = 0x0041
R10 = 0xF000
R11 = 0x0000
R12 = 0x0000
```

```
R13 = 0x0000
R14 = 0x2000
R15 = 0x000B
Z = 0
C = 0
[0x000B] = 0xF000
```

4.3.5 Exemplo 5 (Assembly)

Entrada:

```
main:   MOV R0, #char1          // R0 = addr of first char
        ADDI R1, R15, #2        // R1 = PC + 2 (Ret Addr)
        PUSH R1                 // Push Return Address
        JMP puts                // Call function puts
        HALT                    // Stops CPU!
puts:   PUSH R1                 // Save R1
start:  PUSH R0                 // Save string pointer
        LDR R0, [R0, #0]        // R0 = char value
        OR R0, R0, R0           // Update Zero Flag
        JEQ end                 // If char == 0, return
        ADDI R1, R15, #2        // R1 = PC + 2 (Ret Addr)
        PUSH R1                 // Push Return Address
        JMP putc                // Call function putc
        POP R0                  // Restore string pointer
        ADDI R0, R0, #1         // Pointer++
        JMP start               // Next character
end:    POP R0                  // Restore R0
        POP R1                  // Restore R1
        POP R15                 // Return (Pop PC)
putc:   PUSH R1                 // Save R1
        MOV R1, #0xF0           // R1 = 0x00F0
        SHL R1, R1, #8          // R1 = 0xF000 (IO Base)
        STR R0, [R1, #1]        // Write char to Output
        POP R1                  // Restore R1
        POP R15                 // Return
char1:  'H'
char2:  'e'
char3:  'l'
char4:  'l'
char5:  'o'
char6:  '\0'
```

Saída:

```
OUT <= H
OUT <= e
OUT <= l
OUT <= l
OUT <= o
R0 = 0x001E
```

```
R1 = 0x0004
R2 = 0x0000
R3 = 0x0000
R4 = 0x0000
R5 = 0x0000
R6 = 0x0000
R7 = 0x0000
R8 = 0x0000
R9 = 0x0000
R10 = 0x0000
R11 = 0x0000
R12 = 0x0000
R13 = 0x0000
R14 = 0x2000
R15 = 0x0005
Z = 1
C = 0
[0x0019] = 0x0048
[0x001A] = 0x0065
[0x001B] = 0x006C
[0x001C] = 0x006C
[0x001D] = 0x006F
[0x001E] = 0x0000
```

5 Dicas de Programação

1. Sinta-se a vontade para utilizar o simulador projetado no laboratório 03 como base.

2. Ao imprimir algum dado, lembre-se que são dados de 16 bits! Portanto, use trate de utilizar o tamanho adequado no printf. Por exemplo, pra imprimir em hexadecimal, coloque o h (de half) próximo ao X (de heXa):

```
printf("R%d = 0x%04hX\n", i, cpu.regs[i]);
```

3. Ao imprimir a pilha, verifique se o SP é diferente de sua posição padrão (fim da memória, isto é, 0x2000). Se sim, imprima do fim até a posição apontada por SP:

```
[0x1FFF] = 0x000A
[0x1FFE] = 0x0014
```

4. Crie uma *struct* para manter os registradores da CPU, bem como a memória de 16KB.

5. Para descobrir quais posições da memória foram acessadas para imprimir, mantenha um *array* com o mesmo tamanho da memória, e marque as posições acessadas com LDR e STR.

6. O PC é modificado em toda instrução, portanto crie uma variável que mantenha seu valor antes de ser modificado, de forma a permitir o uso dos breakpoints.

7. Mantenha *defines* com valores constantes, facilitando a leitura posteriormente.

```
#define OP_JMP      (0)
#define OP_JCOND    (1)
#define OP_LDR      (2)
#define OP_STR      (3)
...
```

8. Ao decodificar algumas instruções será necessário realizar a **extensão de sinal** do imediato. Para isso, converta o valor para um tipo com sinal (*signed*) e desloque o valor para a esquerda (para posicionar o bit de sinal do imediato no bit 15) e para a direita (para fazer a extensão de sinal). Por exemplo, a instrução JEQ #-6...

```
0011 1111 1010 0001
1111 1110 1000 0100 (ir << 2)
1111 1111 1111 1010 (ir >> 6)
```

No mais, boa sorte a todos!