

Escola Politécnica da Universidade de São Paulo



PCS3422 – Organização e Arquitetura de Computadores II

Toolchain para LEGv8

Prof. Bruno Albertini

Grupo D

Amanda Parizoto Gales	9853356
Daniel Lavedonio de Lima	8992882
Luiz Victorio Martins	9349311
Rodrigo Yugi Sakai	9351183

SUMÁRIO

1. INTRODUÇÃO	2
2. PROJETO	2
2.1. Toolchain aarch64	2
2.2. Conversor ARMv8 para LEGv8	3
2.3. Transformação do arquivo convertido LEGv8 para binário	4
2.4. Conversor do formato ELF para o formato MIF	5
3. COMO USAR	8
4. TESTES	9
4.1. Teste "Hello World"	9
4.2. Teste Fibonacci e Fatorial	10
5. CONCLUSÃO	11
6. BIBLIOGRAFIA	12

1. INTRODUÇÃO

Este projeto faz parte de um conjunto maior, em que todos os projetos realizados pelos diferentes grupos devem se integrar para a construção de um computador funcional capaz de rodar Linux.

Dado o desafio, ficou ao cargo deste projeto de buscar entender melhor o funcionamento e construir uma toolchain para converter programas de alto nível em código objeto que poderia ser executado nos projetos realizados pelos outros grupos em conjunto.

O resultado disso se encontra no decorrer deste relatório.

2. PROJETO

De acordo com a divisão de temas realizada em sala de aula, o objetivo do projeto em questão é o desenvolvimento de um compilador completo que seja compatível com o conjunto de instruções LEGv8, subset de instruções do ARMv8, tendo como resultado final um arquivo .MIF.

Utilizando-se do conceito de Toolchain: conjunto de ferramentas de compilação, foram desenvolvidas algumas etapas para que se pudesse cumprir totalmente com a proposta do trabalho. As etapas foram divididas em utilizar o compilador aarch64, realizar a conversão das instruções ARMv8 para LEGv8, transformar o arquivo assembly em binário e, por último, converter o formato ELF da saída anterior em binário para o formato MIF, que pode ser lido pelos projetos dos outros grupos.

2.1. Toolchain aarch64

Primeiramente, para tal, fez-se uso de um cross-compiling toolchain chamado **aarch64** em uma máquina virtual Linux (Ubuntu). Optou-se pelo uso de Ubuntu como sistema operacional desse projeto devido a compatibilidades intrínsecas com as ferramentas

usadas nesse projeto, como gcc, python e o próprio aarch64. O download do toolchain foi realizado pelo terminal através do seguinte comando:

```
sudo apt-get install gcc-aarch64-linux-gnu
```

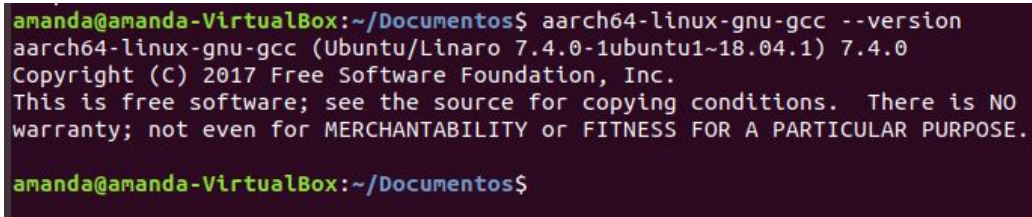
A terminal window with a dark background and light-colored text. The prompt is 'amanda@amanda-VirtualBox:~/Documentos\$'. The command entered is 'aarch64-linux-gnu-gcc --version'. The output shows the version '7.4.0' and copyright information for the Free Software Foundation, Inc. The prompt returns to 'amanda@amanda-VirtualBox:~/Documentos\$'.

Figura 2.1.1 - Versão Aarch64 e Ubuntu

De posse do toolchain em questão instalado, basta compilar o programa desejado, em linguagem C, de forma a gerar o arquivo em assembly (extensão .s) para ARM. Supondo que o nome do arquivo em C seja “prog.c”, utiliza-se o comando `aarch64-linux-gnu-gcc -S -o cod.txt prog.c` para salvar o código assembly em um arquivo txt.

2.2. Conversor ARMv8 para LEGv8

A próxima etapa foi a conversão das instruções ARMv8 para LEGv8. Para tanto buscou-se online pela documentação de todas as instruções em ARMv8 e tentou-se obter para cada uma delas um conjunto de instruções LEGv8 equivalentes. Vale a pena mencionar que, para essa etapa, foram consideradas operações válidas LEGv8 todas as disponíveis menos as de ponto flutuante.

Muitas das operações foram convertidas com sucesso, sem perda de funcionalidade, no entanto houveram casos onde o mesmo não foi possível. Divide-se esses casos em dois tipos específicos de operação: operações substituíveis e não-substituíveis. As operações substituíveis foram aquelas cuja função pode ser substituída por outra LEGv8 de modo a garantir o funcionamento do programa em troca de perda de parte de sua funcionalidade original. Exemplos desse tipo de operação são PRFM, que da pre-fetch em uma região da memória para o cache, e LDNP que realiza o load de duas palavras da memória consecutivas com um “non-temporal hint”. Nesse tipo de caso substitui-se a instrução por uma de caráter parecido e continuou-se a compilação. O outro caso são os de instruções não-substituíveis, cuja funcionalidade não pode ser replicada dentro do conjunto limitado de operações LEGv8,

seja por falta do número registradores necessários na instrução original para a realização da conversão adequada ou pela falta de integração do LEGv8 com o sistema operacional. Alguns exemplos desse tipo de instrução são instruções SIMD, chamadas do sistema operacional e operações de shift lógico variável. Nesses casos é impossível realizar uma conversão funcional, e a compilação, mesmo que rode sem erros, não gerará um código assembly LEGv8 executável.

Estabelecida a conversão de cada operação possível, desenvolveu-se um programa em python 3.7 responsável por realizar a conversão, varrendo o assembly ARM obtido na primeira etapa e gerando um arquivo `saida.s` com um código LEGv8 equivalente.

```
def convert_1_to_3(x):
    if x[0] == 'sbc':
        x1 = ['add', x[1], x[2], 'HS']
        x2 = ['sub', x[1], x[1], x[3]]
        x3 = ['subi', x[1], x[1], '#1']
    elif x[0] == 'sbcs':
        x1 = ['add', x[1], x[2], 'HS']
        x2 = ['sub', x[1], x[1], x[3]]
        x3 = ['subi', x[1], x[1], '#1']
    elif x[0] == 'ngc':
        x1 = ['add', x[1], 'WZR', 'HS']
        x2 = ['sub', x[1], x[1], x[2]]
        x3 = ['subi', x[1], x[1], '#1']
    elif x[0] == 'ngcs':
        x1 = ['add', x[1], 'WZR', 'HS']
        x2 = ['subi', x[1], x[1], '#1']
        x3 = ['subs', x[1], x[1], x[2]]
```

Figura 2.2.1 - Trecho de código do conversor Python

2.3. Transformação do arquivo convertido LEGv8 para binário

A conversão do código LEGv8 para binário se dá mais uma vez através da toolchain `aarch64`. Como LEGv8 é um subconjunto de instruções existentes do ARMv8, todas as suas instruções são compreendidas pelo `aarch64`, bastando o uso da instrução `aarch64-linux-gnu-gcc -c saida.s`, sendo `saida.s` o nome da saída do tradutor python da etapa passada, para a geração do binário em formato ELF (arquivo resultante `saida.o`) necessário para a próxima etapa, a geração do arquivo MIF. Vale notar que caso haja a ocorrência de instruções não-conversíveis no código ARM que não tenham sido detectadas na etapa passada essa etapa falhará.

2.4. Conversor do formato ELF para o formato MIF

A partir do arquivo gerado na etapa anterior, é feita a conversão do formato ELF para o MIF por meio do programa `mif_converter.py`, que pode ser executado com qualquer versão de Python, seja 2.x ou 3.x. Para entender como é feita esta conversão, é necessário entender como estão ordenados ambos os formatos, de origem e destino.

O formato ELF (*Executable and Linkable Format*) é muito usado em diversos sistemas operacionais Unix e Unix-like para a execução de códigos-objeto, cuja grande vantagem é permitir a execução em várias plataformas de hardware diferentes, principalmente pelo suporte de diferentes endianness e tamanhos de endereços de memória. Cada arquivo em formato ELF possui uma seção de headers seguido pelos dados contidos no restante do arquivo. Cada header possui um conjunto de configurações do hardware em que foi codificado e a codificação em que os dados do arquivo se encontram. É possível que o segundo ou o terceiro header não estejam presentes no arquivo final por não conterem informações relevantes para a execução daquele programa e, assim, economizar espaço. Cada header possui diversas configurações; as relevantes estão descritas a seguir:

1. **File Header:** Este header define os parâmetros principais para a leitura do arquivo em si e as posições dos outros headers.
 - a. **e_ident[EI_MAG0] até e_ident[EI_MAG3]:** 0x7F seguido por ELF (45 4c 46) em ASCII; esses quatro bytes constituem o número mágico (se possuírem um código diferente deste, não é um arquivo ELF).
 - b. **e_ident[EI_CLASS]:** Esse byte é definido como 1 ou 2 para indicar a formatação de 32 ou 64 bits, respectivamente.
 - c. **e_ident[EI_DATA]:** Esse byte é definido como 1 ou 2 para significar *little* ou *big endianness*, respectivamente. Isso afeta a interpretação dos campos de bytes múltiplos, indicando se os valores que o compõem devem ser lidos de trás pra frente (*little*) ou não (*big*).

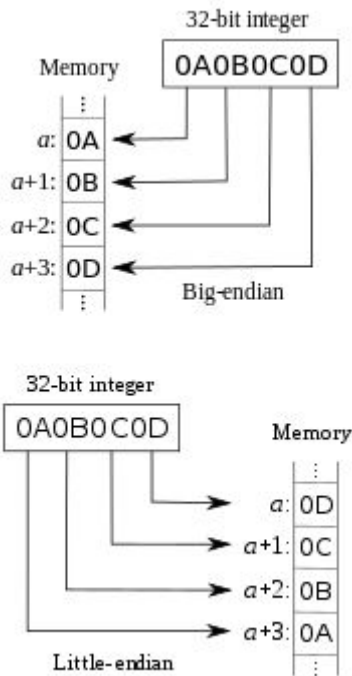


Figura 2.4.1 - Visualização dos conceitos de *little* e *big endiannesses*

- d. **e_ident[EI_VERSION]:** Definido como 1 para a versão original e atual do ELF.
- e. **e_machine:** Especifica a arquitetura do *instruction set* de destino. No caso deste projeto, será o AArch64.
- f. **e_entry:** Este é o endereço de memória do ponto de entrada de onde o processo começa a executar. Este campo tem 32 ou 64 bits, dependendo da formatação definido anteriormente.
- g. **e_phoff:** Aponta para o início do Program Header. Se ele não estiver presente, recebe o valor 0.
- h. **e_shoff:** Aponta para o início do Section Header. Se ele não estiver presente, recebe o valor 0.
- i. **e_ehsize:** Contém o tamanho do File Header.
- j. **e_phentsize:** Contém o tamanho do Program Header. Se ele não estiver presente, recebe o valor 0.
- k. **e_phnum:** Contém o número de entradas no Program Header. Se ele não estiver presente, recebe o valor 0.

- l. **e_shentsize:** Contém o tamanho do Section Header. Se ele não estiver presente, recebe o valor 0.
 - m. **e_shnum:** Contém o número de entradas no Section Header. Se ele não estiver presente, recebe o valor 0.
2. **Program Header:** Este header informa ao sistema como criar uma imagem de processo. Ele é encontrado no deslocamento de arquivo *e_phoff* e consiste em entradas *e_phnum*, cada uma com o tamanho *e_phentsize*.
 3. **Section Header**

O formato MIF (Memory Initialization File) é um arquivo de texto ASCII (com a extensão .mif) que especifica o conteúdo inicial de um bloco de memória (CAM, RAM ou ROM), ou seja, os valores iniciais de cada endereço. Este arquivo é usado durante a compilação e/ou simulação do projeto. Em um MIF, é preciso especificar os valores de profundidade e largura da memória. Além disso, se vários valores forem especificados para o mesmo endereço, apenas o último valor será usado. Um exemplo de arquivo com essa especificação segue abaixo:

```
DEPTH = 32;           % Memory depth and width are required %
                     % DEPTH is the number of addresses      %
WIDTH = 14;           % WIDTH is the number of bits of data per word %
% DEPTH and WIDTH should be entered as decimal numbers      %

ADDRESS_RADIX = HEX;  % Address and value radices are required %
DATA_RADIX = HEX;     % Enter BIN, DEC, HEX, OCT, or UNS; unless %
                     % otherwise specified, radices = HEX      %

-- Specify values for addresses, which can be single address or range
CONTENT
BEGIN
[0..F]: 3FFF; % Range--Every address from 0 to F = 3FFF %
6      : F;  % Single address--Address 6 = F %
8      : F E 5; % Range starting from specific address %
--      % Addr[8] = F, Addr[9] = E, Addr[A] = 5 %
END;
```

Figura 2.4.2 - Exemplo de arquivo com formato MIF

O conversor em si funciona utilizando um dicionário para coletar as informações presentes nos *headers* do arquivo em formato ELF e utilizá-las para servir de parâmetro para a leitura correta dos dados do arquivo (*little e big endianness*, por exemplo, afetam bastante como ele está escrito). Quando um header não está presente na estrutura do arquivo, seu

tamanho e posição inicial estão setados como zero e são ignorados. Uma vez que os headers são totalmente lidos e os dados do arquivo lidos de forma correta, o programa passa a escrever um arquivo MIF, com estrutura similar a apresentada anteriormente, com os parâmetros contidos no header (principalmente *DEPTH* e *WIDTH*) e os dados armazenados contidos no arquivo ELF. Finalizado isso, o arquivo de saída é salvo como o mesmo nome do de entrada, só que com extensão **.mif**.

3. COMO USAR

Primeiramente deve-se instalar o toolchain aarch64 em um ambiente Linux. Se estiver utilizando o Ubuntu, deve-se utilizar o comando:

```
sudo apt-get install gcc-aarch64-linux-gnu
```

Uma vez instalado, deve-se compilar o programa desejado (neste caso o `file.c`) para gerar um arquivo assembly para ARM com o seguinte comando:

```
aarch64-linux-gnu-gcc -S -o ARM_assembly.txt file.c
```

Em seguida deve-se converter o formato assembly de ARM para o formato assembly LEGv8. Utiliza-se o programa `arm_to_v8.py`, com o qual escolhe-se o arquivo a ser utilizado, com o seguinte comando:

```
python arm_to_v8.py
```

A saída desse programa é um arquivo `.txt`, portanto deve-se renomeá-lo para `.s`. Então, converte-se o arquivo assembly resultante da conversão anterior (neste caso o `file.s`) em formato objeto (`.o`, formato ELF) com a toolchain aarch64, a partir do comando:

```
aarch64-linux-gnu-gcc -c file.s
```

Por último, o arquivo em formato objeto `.o` deve ser convertido para formato MIF (`.mif`), usando o programa `mif_converter.py`, com o seguinte comando:

```
python mif_converter.py file.o
```

O resultado final deve ser um arquivo *file.mif* totalmente convertido para instruções LEGv8 e pronto para ser executado em memória a partir do programa Quartus.

4. TESTES

4.1. Teste "Hello World"

Neste teste, foi criado um programa em C contendo apenas um *printf* da string "Hello World", conforme mostrado abaixo:

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Figura 4.1.1 - Programa teste em C

De posse do programa em assembly em um arquivo *.txt*, executa-se o código para gerar outro arquivo *.txt* com o código correspondente para o LEGv8.

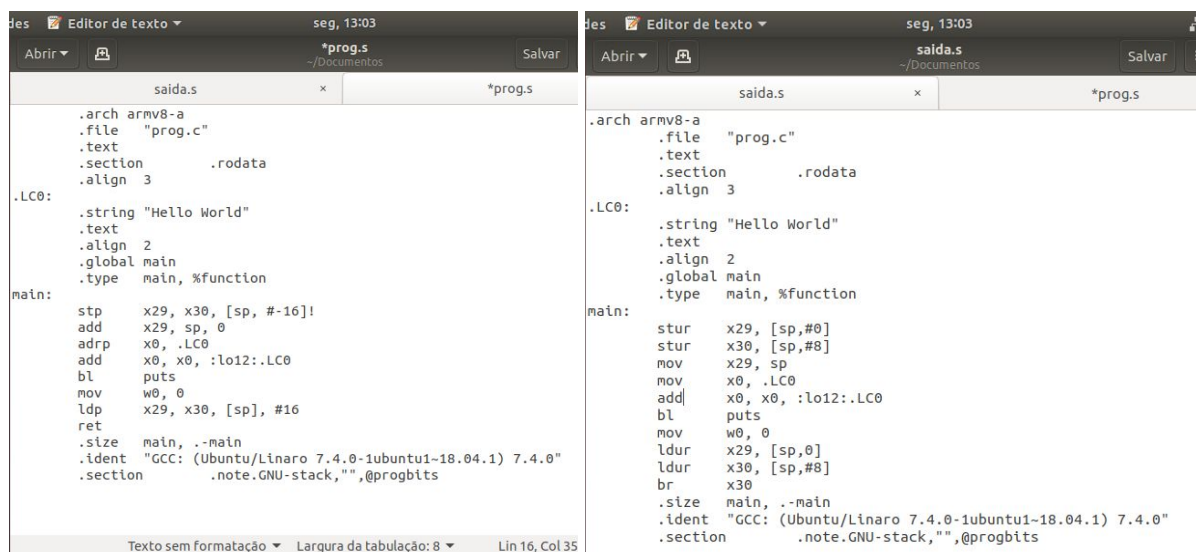
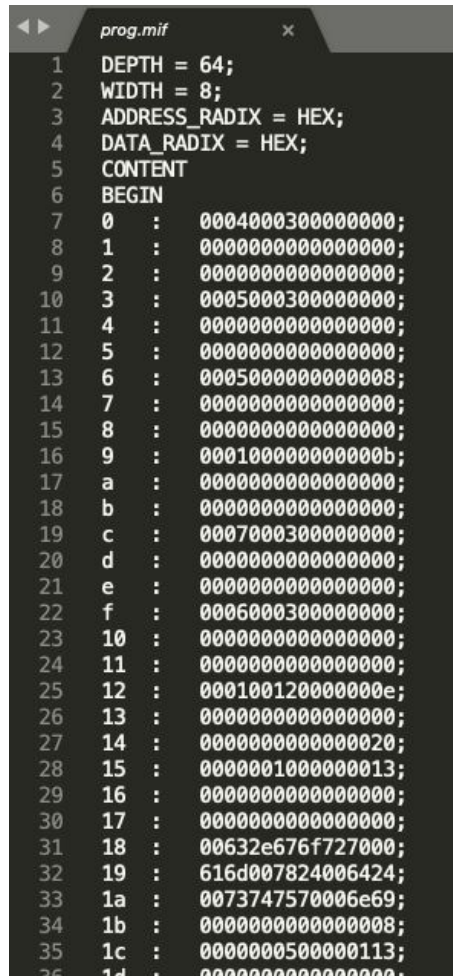


Figura 4.1.2 - Programa teste em assembly para ARM e LEGv8 respectivamente

Então, o arquivo .s referente às instruções para o LEGv8 é utilizado no programa de conversão para MIF, com o comando explicitado acima. O resultado final pode ser conferido na figura a seguir.



```
prog.mif
1  DEPTH = 64;
2  WIDTH = 8;
3  ADDRESS_RADIX = HEX;
4  DATA_RADIX = HEX;
5  CONTENT
6  BEGIN
7  0   : 0004000300000000;
8  1   : 0000000000000000;
9  2   : 0000000000000000;
10  3   : 0005000300000000;
11  4   : 0000000000000000;
12  5   : 0000000000000000;
13  6   : 0005000000000008;
14  7   : 0000000000000000;
15  8   : 0000000000000000;
16  9   : 000100000000000b;
17  a   : 0000000000000000;
18  b   : 0000000000000000;
19  c   : 0007000300000000;
20  d   : 0000000000000000;
21  e   : 0000000000000000;
22  f   : 0006000300000000;
23  10  : 0000000000000000;
24  11  : 0000000000000000;
25  12  : 000100120000000e;
26  13  : 0000000000000000;
27  14  : 0000000000000020;
28  15  : 0000001000000013;
29  16  : 0000000000000000;
30  17  : 0000000000000000;
31  18  : 00632e676f727000;
32  19  : 616d007824006424;
33  1a  : 0073747570006e69;
34  1b  : 0000000000000008;
35  1c  : 0000000500000113;
36  1d  : 0000000000000000;
```

Figura 4.1.3 - Arquivo final em formato MIF

4.2. Teste Fatorial

Neste teste, foi criado um programa em C que realizasse o cálculo do enésimo elemento da sequência de Fibonacci e também o seu valor fatorial. O programa em questão está descrito abaixo:

```

1  #include <stdio.h>
2
3  int factorial (int n) {
4      if(n == 0) {
5          return 1;
6      }
7      else {
8          return n * factorial(n-1);
9      }
10 }
11
12 int main () {
13     int n = 4;
14     int fact = factorial(n);
15     return 0;
16 }

```

Figura 4.2.1 - Programa em C do cálculo de fatorial

Da mesma forma que para o teste anterior, utiliza-se os comandos de forma a obter o código assembly em .txt para o ARM. Nesse caso, como o assembly tem mais de um bloco de instruções (.L2, .L3, main e factorial), recomenda-se salvar o bloco referente a *main* em outro arquivo de modo a garantir que o programa *arm_to_leg.py* funcione melhor.

```

factorial:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    w0, [x29, 28]
    ldr    w0, [x29, 28]
    cmp    w0, 0
    bne    .L2
    mov    w0, 1
    b      .L3
.L2:
    ldr    w0, [x29, 28]
    sub    w0, w0, #1
    bl     factorial
    mov    w1, w0
    ldr    w0, [x29, 28]
    mul    w0, w1, w0
.L3:
    ldp    x29, x30, [sp], 32
    ret
.size    factorial, .-factorial
.align   2
.global  main
.type    main, %function
main:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    mov    w0, 4
    str    w0, [x29, 24]
    ldr    w0, [x29, 24]
    .
    .
    .

```

```

factorial:
    stp x29, x30, [sp, -32]!
    add x29, sp, 0
    ldur w0, [x29, 28]
    bne .L2
    mov w0, 1
    b .L3
.L2:
    ldur w0, [x29, 28]
    sub w0, w0, #1
    bl factorial
    mov w1, w0
    ldur w0, [x29, 28]
    mul w0, w1, w0
.L3:
    ldur x29, [sp,#0]
    ldur x30, [sp,#8]
    br x30
.size factorial, .-factorial
.align 2
.global main
.type main, %function
main:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    mov    w0, 4
    ldur    w0, [x29, 24]
    bl     factorial
    mov    w0, 0
    ldur    x29, [sp,#0]
    ldur    x30, [sp,#8]

```

Figura 4.2.2 - Códigos assembly para ARM e LEGv8 respectivamente

Em posse do arquivo .s com pequenas correções de espaçamento e “[]”, executa-se o comando para obter o arquivo ELF e, por fim, tem-se o MIF através do segundo programa python.

5. CONCLUSÃO

Esse projeto possibilitou a melhor compreensão do funcionamento de compiladores e, principalmente, de como a linguagem de baixo nível se relaciona com sua linguagem de máquina e com o processador utilizado. Apesar de serem numerosas e requererem grande trabalho, também estudou-se a respeito das instruções utilizadas em ARM e como se diferem em relação ao LEGv8. Por fim, também houve aprofundamento do conhecimento a respeito dos arquivos tipo MIF.

6. BIBLIOGRAFIA

<http://www.eecs.umich.edu/courses/eecs370/eecs370.f19/resources/materials/ARM-v8-Quick-Reference-Guide.pdf>

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802a/ADRP.html>

https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[https://en.wikipedia.org/wiki/Magic_number_\(programming\)](https://en.wikipedia.org/wiki/Magic_number_(programming))

<https://en.wikipedia.org/wiki/Endianness>

https://www.mil.ufl.edu/4712/docs/mif_help.pdf

https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_mif.htm