Bruno Farias de Loreto

PORTANDO O EPOS PARA A ZEDBOARD

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador
Universidade Federal de Santa Catarina: Prof. Dr. Antônio Augusto Fröhlich
Coorientador
Instituto Federal de Santa Catarina: Prof. Dr. Giovani Gracioli

Florianópolis

2014

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Loreto, Bruno
Portanto o EPOS para a Zedboard / Bruno Loreto ; orientador, Antônio Augusto Fröhlich ; coorientador, Giovani Gracioli. - Florianópolis, SC, 2014.
60 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Graduação em Ciências da Computação.

Inclui referências

1. Ciências da Computação. 2. Sistemas Operacionais. 3. Portabilidade. 4. EPOS. I. Augusto Fröhlich, Antônio. II. Gracioli, Giovani. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Bruno Farias de Loreto

PORTANDO O EPOS PARA A ZEDBOARD

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pela Universidade Federal de Santa Catarina.

Florianópolis, 12 de Novembro 2014.

Prof. Dr. Vitório Bruno Mazzola
Coordenador
Universidade Federal de Santa Catarina

Banca Examinadora:

Prof. Dr. Antônio Augusto Fröhlich
Orientador
Universidade Federal de Santa Catarina

Prof. Dr. Giovani Gracioli
Coorientador
Instituto Federal de Santa Catarina

Prof. Me. Arliones Stevert Hoeller Junior
Instituto Federal de Santa Catarina

Prof. Me. Hugo Marcondes
Instituto Federal de Santa Catarina

AGRADECIMENTOS

Agradeço ao Goku por ter derrotado o Freeza, aos resumos da Thaís, e às pessoas que jogaram Magic comigo. Agradeço também aos happy hours, pois sem eles eu teria me formado cedo demais.

Faço um agradecimento especial à todos os UFSCães, simplesmente por existirem, e ao RU por me manter vivo. Eu também não poderia esquecer de agradecer ao stackoverflow, à todas as pessoas que fazem tutoriais na internet, e a todos os adoradores do software livre.

Finalmente, um beijo no coração para todos aqueles que desejam falir a Microsoft, e dois beijos para quem não vota no PSDB.

So long, and thanks for all the fish!
- *Douglas Adams.*

RESUMO

Com o aumento da demanda de poder de processamento dos sistemas embarcados atuais, tornou-se necessário que um sistema operacional embarcado tenha suporte para arquiteturas multicores embarcadas. Um sistema operacionanal embarcado tendo suporte para tal plataforma, possibilita novas linhas de pesquisa, e novos cenários de aplicação do SO, como por exemplo a integração de um escalonador *multicore* de tempo real para gerenciar tarefas críticas. O sistema operacional EPOS não possui suporte para um arquitetura embarcada *multicore*, e tem sido cada vez mais necessário a existência de tal suporte. O EPOS possui uma arquitetura que tenta ser o mais independente de plataforma possível, entretanto o interfaceamento entre o software e hardware inevitavelmente necessita ser reescrito para cara arquitetura. Este trabalho visa descrever e documentar como o sistema operacional EPOS foi portado para a plataforma embarcada *multicore* Zedboard.

Palavras-chave: Sistemas Operacionais. Portabilidade. EPOS.

ABSTRACT

With the increasing demand for processing power from nowadays embedded systems, it became necessary for an embedded operating system to support multicore platforms. A embedded operating system having support for such platform enables new research lines, as well as new deployment scenarios, such as the integration of a multicore real time scheduler to manage critical tasks. The EPOS operating system does not have support for a multicore embedded platform, and such support is becoming increasingly necessary. EPOS's operating system design aims on portability, however the software-hardware interface inevitably must be rewritten for each platform. This work aims on describing and documenting how the operating system EPOS was ported to the multicore embedded platform Zedboard.

Keywords: Operating Systems. Porting. EPOS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SoC *System-on-Chip*. 21

TCM *Tightly Coupled Memory* 21

MMU *Memory Management Unit* 21

SP *Stack Pointer* 21

PC *Program Counter* 21

LR *Link Register* 21

CPSR *Current Program Status Register*. O registrador que armazena o atual status do processador operante. 21

SPSR *Saved Program Status Register*. (Registrador que armazena o valor de CPSR no momento imediatamente anterior ao acontecimento de uma exceção, deste modo o antigo valor de CPSR pode ser restaurado quando a exceção for tratada. 21

CP15 *System Control Coprocessor* 21

IRQ Interrupção normal. 21

FIQ *Fast Interrupt*. 21

APSR *Application Program Status Register*. Armazena uma cópia do estado das *flags* da unidade lógico-aritimética. Conhecido também como *flag* do código condicional, usados para determinar se uma instrução condicional deve ser executada ou não. 21

GIC *Generic Interrupt Controler* 21

rx Quando a abreviação rx aparecer, ela estará se referindo genericamente à qualquer um dos 13 primeiros registradores de propósito geral do ARM9 [r0-r12]. 21

PPI *Private Peripheral Interrupt.* 21

SGI *Software Generated Interrupt.* 21

SPI *Shared Peripheral Interrupt.* 21

PS *Processing system.* 21

PL *Programmable Logic.* 21

SMC *Static Memory Controller* 21

TSC *Time Stamp Counter.* 21

FPGA *Field-Programmable Gate Array*. Hardware reconfigurável, o qual têm as suas funcionalidades definidas exclusivamente pelos usuários. 21

PLL *Phase-Locked Loop*. É um sistema de controle que gera uma

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados estão instalados e controlam os mais diversos equipamentos hoje em dia, eles estão mais presentes do que nos damos conta. Conforme eles estão sendo cada vez mais empregados, sua demanda aumenta também, bem como seu poder de processamento. Máquinas que antes utilizavam vários pequenos microcontroladores trabalhando independentemente, hoje podem ser gerenciados com melhor custo-benefício por um sistema de maior desempenho (Gracioli, 2014).

Sistemas embarcados estão empregados em tarefas que exigem a manipulação de grande volume de dados (muitas vezes em tempo real), como na área de comunicação e transmissão de dados. O nome "sistema embarcado" já foi sinônimo de pequenos microcontroladores dedicados, mas hoje também há grande demanda por sistemas embarcados de alto desempenho. Um meio de obter processamento de alto desempenho à baixo custo é com a implantação de processadores *multicores*.

Aplicações de tempo real necessitam um tempo de resposta previsível, e suas tarefas devem ser executadas com atraso controlável. Um exemplo são os sistemas de airbag automotivo, que necessitam de uma resposta a um estímulo em um determinado intervalo de tempo (NATIONAL. . . , 2014). Muitas destas aplicações são feitas diretamente em hardware para se adequar aos requisitos da aplicação (tempo real, gasto energético, etc). Entretanto aplicações feitas diretamente em hardware apresentam problemas em relação à manutenibilidade e possibilidade de atualização (Gracioli, 2014). Estes componentes de hardware poderiam ser centralizados em um sistema embarcado de maior processamento, e alocar tarefas de tempo real para cada uma das funções necessárias ao sistema. Deste modo, a manutenção e custo total do sistema pode ser melhorado, sem necessitar sacrificar as restrições temporais (usando um sistema operacional de tempo real que consiga escalonar o conjunto de tarefas).

Outro exemplo são os quadcópteros. Controlar um quadcóptero não é uma tarefa trivial, há uma miríade de fatores a serem considerados. Um quadcóptero precisa balancear quanto empuxo deve ser aplicado à cada hélice para manter-se no ar e fazer manobras, é necessário também que o sistema seja tolerante à faltas para ser seguro. Em caso de um quadcóptero autônomo, os desafios são ainda maiores. Passa a ser necessário geolocalização, detectar à que distância se está do chão, controle de pouso, etc (Lee; Seshia, 2011). Não é difícil perceber a grande quantidade de dados a serem processados, e que uma falha em processar estes dados no tempo estimado pode gerar numa falha total do sistema (a queda do quadcóptero, neste caso). Neste exemplo,

nota-se como é necessário um sistema operacional previsível, e um hardware capaz de processar rapidamente e paralelamente diversos dados.

É natural que estas plataformas de maior desempenho, que consequentemente possuem mais recursos e componentes à serem gerenciados, necessitem de um sistema operacional que se encarregue desta função, e que possam administrar eficientemente estes recursos. De acordo com um estudo recente (UBM, 2013), a principal razão para usar um sistema operacional é ter garantias de tempo real.

De acordo com este mesmo estudo (UBM, 2013), metade dos projetos de sistemas embarcados usaram mais que um microcontrolador/processador em seu projeto. Além disto, tem sido comum o uso de FPGAs[1] integradas à estes sistemas, onde 31% dos projetos usaram FPGAs em 2013. Dentre estas placas, podemos destacar a Zedboard, que é uma plataforma de bom desempenho[2] com um processador *dualcore* de 666 MHz e FPGA.

Uma peça central de um sistema operacional de tempo real é seu escalonador de processos/*threads*. É neste componente que políticas sobre priorização de tarefas são feitas de modo a garantir que cada tarefa seja cumprida dentro de seu prazo.

EPOS, que é um sistema operacional embarcado de tempo real (RTOS[3]) *multithread*, é o o primeiro RTOS de código aberto a suportar os escalonadores de tempo real global, particionado e agrupado (Gracioli, 2014).

Estes fatos demonstram a necessidade de um sistema operacional de tempo real como o EPOS de ter suporte para uma plataforma embarcada multicore. A única arquitetura *multicore* suportada pelo EPOS é a IA32, que não é uma arquitetura que oferece previsibilidade, logo para todos os sistemas embarcados com restrições temporais uma arquitetura multicore com maior previsibilidade torna-se necessário.

O EPOS não possuia suporte para um processador embarcado *multicore*. Este trabalho trata de descrever como foi feito o porte [4] deste sistema operacional para a Zedboard, uma plataforma embarcada *multicore*.

---

[1] FPGAs podem ser pensadas como um hardware que pode ser programado pelo usuário.

[2] Comparado ao poder de processamento dos demais processadores embarcados.

[3] Real-Time Operating System

[4] Porte é um estrangeirismo da palavra *port*, que, no âmbito da computação, significa o ato de fazer um mesmo programa/sistema/SO funcionar em diferentes ambientes. Por exemplo, fazer um software que antes só funcionava no Linux passar a funcionar em um outro SO (sistema operacional) pode ser considerado um porte. Uma palavra alternativa que poderia ser usada é "suporte", entretanto acredito que esta palavra não expresse apropriadamente o que foi feito, já que essa palavra normalmente é associada com um serviço pago de assistência técnica, e o próprio fraseamento do que foi feito se tornaria de mais difícil compreensão e prolixo com esta palavra.

1.1 OBJETIVOS GERAIS

O objetivo deste trabalho é portar o sistema operacional EPOS para a Zedboard e documentar este processo no presente documento. O porte consiste em adaptar os componentes que, dentro da arquitetura do EPOS, são chamados de mediadores de hardware. Cada mediador precisa ser refeito para cada plataforma[5].

1.2 OBJETIVOS ESPECÍFICOS

Dos objetivos específicos, pode-se organizá-los da seguinte forma:

1. Estudo da arquitetura do EPOS, da arquitetura do ARM Cortex A9 e do Zedboard.
2. Portes
   (a) UART
   (b) Timers
   (c) Controlador de Interrupção
   (d) MMU
   (e) CPU
   (f) Inicialização *multicore*
3. Documentação

1.3 ORGANIZAÇÃO DO DOCUMENTO

O restante deste documento é organizado da seguinte forma: No capílo de conceitos básicos serão discutidos de forma suscinta conceitos usados ao longo do trabalho; no capítulo seguinte será dado uma visão geral do hardware escolhido para o porte; no capítulo 4 será discutido a arquitetura do EPOS, sob um ponto de vista de mais alto nível, sem ainda chegar nos mediadores; e, finalmente, no capítulo 5, será explicado o porte em si, comentando as decisões que foram tomadas e como foi organizado o porte[6].

[5] Aqui chamamos de plataforma qualquer SoC (*System On-Chip*), ou processador. No capítulo 4 será explicado a diferença entre *machine* e *architecture*, e por enquanto estamos usando a palavra plataforma para nos referenciarmos a estes dois conceitos simultaneamente.

[6] Organização da memória, sequência de inicialização, etc.

2 CONCEITOS BÁSICOS

Esta seção tem por objetivo o estabelecimento dos conceitos básicos necessários ao entendimento do restante do trabalho. Não pretende-se aqui fazer uma explicação extensiva e profunda sobre os conceitos expostos, mas sim uma breve introdução, deixando os detalhes para serem procurados pelo leitor nas respectivas referências.

2.1 SISTEMA OPERACIONAL

Tanenbaum define sistemas operacionais sob duas perspectivas (Tanenbaum, 1997):

SO como uma máquina extendida: Programar ao nível de linguagem de máquina pode ser um tanto desastroso e desnecessariamente complicado. Por exemplo, na arquitetura PD765, o controlador de um disquete possui 16 comandos de controle, sendo os mais básicos os comandos WRITE e READ. Cada um destes comandos requerem 13 parâmetros, codificados em 9 bytes, parâmetros estes que específicam o endereço do bloco a ser lido, número de setores por trilha, modo de gravação desejada, etc.

Certamente um programador não quer ter que lidar com esse tipo de detalhe, e no lugar ele gostaria apenas de ter uma simples abstração de arquivos que podem ser abertos, editados, salvos e fechados. O programa que procura esconder estes detalhes sórdidos é o sistema operacional, que procura abstrair o gerenciamento de arquivos, assim como detalhes sobre tratamento de interrupções, timers, gerenciamento de memória e qualquer outro detalhe de baixo nível, fazendo assim o programador ter a ilusão de estar interagindo com uma máquina mais simples e de mais alto nível.

SO como um gerenciador de recursos: Diferentes processos competem entre si por utilizar os recursos do sistema. Estes processos precisam compartilhar entre si memória, tempo na CPU, dispositivos de entrada e saída, impressoras, etc. O programa responsável por conciliar as necessidades de cada processo, gerenciando os recursos da máquina de forma organizada e que atenda à todas as requisições por eles é, também, o sistema operacional.

2.1.1 Processo

Um processo basicamente é um programa *em execução*, incluindo o seu *program counter*, registradores e variáveis (Tanenbaum, 1997). Cada

processo possui um espaço de endereçamento próprio e individual, estando protegidos (e impedidos) de acessarem a porção de memória alocada para os demais processos. Cada processo executa na CPU por um certo intervalo de tempo, e então, caso não tenha terminado sua execução, esta é interrompida pelo escalonador de processos (veja seção 2.1.3) para que a CPU possa ser usada por outro processo.

2.1.2 Thread

Todo processo está rodando pelo menos uma thread, e um conjunto de threads pode pertencer ao mesmo processo (toda thread pertence à algum processo). Cada thread pode executar independentemente, de forma pseudo-paralela [1] , entretanto cada thread compartilha do espaço de endereçamento do processo ao qual pertencem. Cada thread possui seu próprio contexto, isto é, o estado de seus registradores (incluindo o *program counter*, pilha e etc). A vantagem de threads sobre processos é que a comunicação entre threads de um mesmo processo é muito mais rápida que a comunicação intraprocessos, e criar uma thread é mais rápido que criar um processo também.

2.1.3 Escalonador

Um escalonador de processos (ou threads) é responsável por garantir que cada processo/thread consiga executar por algum determinado tempo na CPU. Existem diversos algoritmos de escalonamento, vários deles tendo sido implementados no EPOS. De acordo com Tanenbaum, há cinco dimensões que um escalonador deve levar em consideração em sua política de escalonamento (Tanenbaum, 1997):

1. **Justiça (Fairness):** Assegurar que cada processo consiga executar por um tempo justo na CPU.

2. **Eficiência:** Tentar manter a CPU ocupada pela maior parte do tempo.

3. **Tempo de Resposta:** Minimizar o tempo de reposta para aplicações interativas.

[1] Em uma arquitetura de processador com apenas um núcleo, todos os processos e threads executam na realidade sequencialmente, entretanto parecem ser executadas paralelamente devido à pequena fatia de tempo que cada processo/thread possui para executar, por isto chamamos isto de pseudoparalelismo. Mesmo em arquiteturas de multiplos núcleos o paralelismo "verdadeiro" ainda não acontece com frequência, já que basta existirem mais processos/threads do que o número de núcleos para que estes tenham que ser sequenciados.

4. **Turnaround:** Minimizar o tempo que processos *batch* precisam esperar para executar.

5. **Vazão (throughput):** Maximizar a quantidade de tarefas realizadas por unidade de tempo.

Claro que alguns destes itens são contraditórios entre si, por isto a escolha de um algoritmo de escalonamento depende da aplicação e carga de trabalho que o sistema pretende lidar.

2.2 SISTEMA EMBARCADO

É cada vez mais difícil definir o que é um sistema embarcado. Hoje sistemas embarcados estão em todos os lugares, eles acionam motores, freios, cintos de segurança, airbags, sistemas de audio em um carro. Eles codificam digitalmente voz e constroem sinais de radio e enviam eles de um celular para uma estação. Eles controlam microondas, ar-condicionados, semáforos, etc.

Existem sistemas embarcados dos mais diversos tamanhos e capacidades, desde aqueles com um *clock* de poucos KHz, até processadores com GHz. Podemos dizer o que *não* é um sistema embarcado: Notebooks, desktop, servidores e supercomputadores. É arriscar incorrer em erro tentar dar uma definição mais precisa (Lee; Seshia, 2011).

2.3 SISTEMAS DE TEMPO REAL

Sistemas de tempo real são definidos como aqueles sistemas nos quais a corretude geral do sistema depende tanto da corretude funcional (produzir o resultado correto) quanto a corretude temporal (produzir o resultado dentro do tempo máximo de tolerância). A corretude temporal é pelo menos tão importante quanto a corretude funcional (Li, 2003). Ou seja, cada tarefa não somente deve produzir o resultado correto, mas também produzi-lo dentro de um intervalo de tempo predeterminado. Note que nem todos os sistemas de tempo real são sistemas embarcados, apesar de existir uma grande área em comum [2](Li, 2003). Os sistemas de tempo real podem ser classificados em duas categorias:

Hard real-time Nesta categoria, as tarefas dos sistemas devem necessariamente ser concluídas dentro do prazo, do contrário a resposta da tarefa

[2]Um sistema que oferece um serviço de *streaming* de vídeo, por exemplo, é um sistema *soft real-time*, mas não necessariamente embarcado.

já não faz mais sentido, e esta perda de prazo pode causar uma falha catastrófica do sistema, possivelmente causando grande prejuízo e até mesmo por vidas humanas em risco.

Soft real-time Estes sistemas também precisam que suas tarefas sejam concluídas dentro do prazo, entretanto há certa tolerancia à atrasos, e no lugar de causar uma falha no sistema, a sucessiva perda de prazos costuma causar degradação do desempenho do sistema.

Exemplos de sistemas *hard real-time* são vários. Imagine um sistema de detecção de mísseis, e que procura abater estes mísseis atirando neles. Fica claro que o tempo entre a detecção e a ação (atirar no míssil) está limitada à uma certa janela de tempo, e que a perda dessa janela de tempo pode resultar em ser acertado por um míssil (Li, 2003).

Por outro lado, podemos imaginar também um reprodutor de vídeo. Cada *frame* deve ser decodificado dentro de certa janela de tempo, do contrário aquele quadro pode não mais fazer sentido. Entretanto estes sistemas são tolerantes à perda de alguns quadros, portanto uma demora na codificação resulta numa degradação no desempenho (menor taxa de quadros por segundo), mas não evitam do vídeo não ser mais reproduzível. Este é um exemplo de sistema *soft real-time*.

2.4 EXCEÇÕES E INTERRUPÇÕES

Na terminologia adotada neste trabalho, uma exceção é um evento inesperado, que interrompe o processando normal do programa, fazendo o registrador PC apontar para uma região específica de memória, onde o tratador daquela exceção reside. Estas exceções incluem erros de permissão de leitura de certa parte da memória, tentativa de execução de uma instrução inválida (mal formada), etc.

Interrupções são bastante semelhantes à exceções. Elas param o fluxo normal de processamento, e também fazem com que a próxima instrução executada seja aquela numa região pré determinada da memória onde está o tratador daquela interrupção. A diferença esta no fato de interrupções poderem ser mascaradas, desativadas e gerenciadas pelo controlador de interrupção, enquanto não há controle sobre a execução das exceções (não são passíveis de desativação). Outra diferença, mais conceitual, é o fato que interrupções em geral são desejáveis e frequentemente necessárias para o funcionamento do sistema, enquanto exceções ocorrem apenas quando algo imprevisto é executado (como a execução de uma área de memória que não possui instruções executáveis).

A princípio, do ponto de vista do processador, uma interrução não tem muita diferença de uma exceção, por isto que no primeiro nível a interrupção é pré processada por um tratador de exceções, para então ser encaminhada para um tratador de interrupções (como mostra a figura 6).

2.5 ADESD

ADESD (*Application-Oriented System Design*) é um método de modelar de sistemas orientados à aplicação, em contrapartida aos sistemas operacionais de propósito geral. Sistemas operacionais de propósito geral criam vários serviços e abstrações que acabam sendo nunca usados pela aplicação, e em alguns casos não oferecendo outras que a aplicação necessitaria (Fröhlich, 2001). Deste modo, no ADESD, o sistema se adapta à aplicação, e não a aplicação que deve se adaptar ao sistema. O EPOS é um sistema operacional criado usando esta perspectiva. Nas subseções seguintes será discutido algumas das técnicas usadas neste modelo, e que são exploradas durante o trabalho. Note que esta discussão é retomada no capítulo 4.

2.5.1 Componente

Um conceito bastante usado em ADESD é a noção de componente, um design baseado em componentes oferece meios para alcançar um SO orientado à aplicação (Fröhlich, 2001, p. 4). Não há um consenso sobre como definir o que é um componente, Grady Booch define como “Um componente reusável de software é um módulo logicamente coeso, fracamente aclopado que denota uma única abstração”, entretanto há muitas outras formas de definir isto.

2.5.2 Metaprogramação estática

Metaprograma é um program que representa e manipula outros programas ou eles próprios (Booch, 1998). Com metaprogramação é possível fazer-se loops (através de recursão), e condicionais (como o `IF` discutido abaixo), fazendo com que metaprogramas tenham equivalência à máquina de turing (Booch, 1998). De fato, a linguagem de metaprogramação do C++ pode ser vista como uma linguagem funcional que tem sua execução em tempo de compilação. Pode-se fazer muitas estruturas metaprogramadas, inclusive listas, árvores e etc, entretanto nesta seção será mostrado apenas o funciona-

mento básico de um `IF` metaprogramado, pois este é bastante usado no código do EPOS. Para um maior entendimento sobre o assunto, fica aqui recomendado o livro referenciado (Booch, 1998).

A arquitetura do EPOS usa com frequência metaprogramação estática para, em tempo de compilação, selecionar a arquitetura, bem como cada componente que será ou não utilizado. Por exemplo, podemos definir um `if` estático para selecionar de qual classe a classe Chronometer irá herdar. Primeiramente a definição do `if`:

```
template<bool condition, typename Then, typename Else>
struct IF
{
    typedef Then Result;
};
```

Este template, a princípio, nada mais faz do que tomar 3 parâmetros e então criar um tipo chamado `Result` igual ao segundo parâmetro, contudo, se fizermos uma especialização deste template, ele passa ser útil:

```
template<typename Then, typename Else>
struct IF<false, Then, Else>
{
    typedef Else Result;
};
```

Com esta especialização, toda vez que o primeiro parâmetro resolver-se como falso, `Result` será definido como o terceiro parâmetro. Temos, portanto, um `if` metaprogramado funcional, agora voltemos ao exemplo do cronômetro.

Suponha que caso a arquitetura não seja multicore, e nos traits o TSC (*time stamp clock*) esteja ativo, então deseja-se que Chronometer herde de TSC_Chronometer, do contrário de Alarm_Chronometer.

Isto pode ser feito usando nosso `if` metaprogramado da seguinte forma:

```
class Chronometer: public
IF<Traits<TSC>::enabled && !Traits<System>::multicore,
TSC_Chronometer, Alarm_Chronometer>::Result
{/*class body*/};
```

Deste modo, `Chronomometer` herdará de `IF::Result`, que será resolvido como TSC_Chronometer ou Alarm_Chronometer. Note que esse exemplo também mostra um uso do Traits, onde para descobrir se o TSC está ativo, bastou ler a constante `Traits<TSC>::enabled`, e, para saber se o sistema é multicore, bastou ler `Traits<System>::multicore`. Todo esse processamento causa zero *overhead* em tempo de execução.

2.5.3 Traits

O uso de traits é uma técnica de metaprogramação estática para associar informações aos objetos em tempo de compilação. Por exemplo, uma matriz metaprogramada pode necessitar saber sua dimensão em tempo de compilação. No EPOS, por exemplo, no momento de criação da heap, e de inicialização da MMU, é necessário obter-se informações a respeito do mapeamento de memória do EPOS; estas informações são obtidas através do uso de traits.

Há basicamente 3 maneiras de definirmos traits (Booch, 1998):

Traits como membro da classe: Forma mais simples de definir-se um trait. Esta técnica basicamente consiste em criar-se constantes e tipos dentro da própria classe que os usa. Esta tecnica é muitas vezes usada sem o programador ter sequer conhecimento que está usando traits.

Classes de traits: Quando um tipo pode ter muitas características, e pode ser conveniente encapsulá-las numa única classe.

Templates de Traits: Consiste em definir uma classe de templates para gerenciar uma família de tipos. Esta é a técnica usada no EPOS, e será discutida melhor abaixo.

Nesta técnica, cria-se uma classe genérica, para que esta seja especializada por cada tipo que precisar de traits. Por exemplo, podemos fazer uma classe genérica `number`, para associar informações sobre tipos numéricos, e então especializar esta classe para os diferentes tipos, como `int`, `long`, etc:

```
template<class T>
class number {
public:
    static const bool specialized = false;
    static const unsigned long max = 0;
    static const unsigned long min = 0;
    static const bool sig = false; //Numero com sinal
};
template<>
class number<int> {
public:
    static const bool specialized = true;
    static const unsigned long max = 2147483647;
    static const long min = -2147483648;
    static const bool sig = true;
};
template<>
class number<unsigned long> {
```

```
public:
    static const bool specialized = true;
    static const unsigned long max = 4294967296;
    static const long min = 0;
    static const bool sig = false;
};
#include<iostream>
int main(){
    //exemplo de uso
    if(number<int>::specialized)
        std::cout << number<int>::min << std::endl;
    else
        std::cout << "Sem informacoes sobre int\n";

    return 0;
}
```

Neste exemplo, poderia-se especializar os demais tipos da linguagem C++, assim como adicionar novos atributos à estas especializações. Note que deste modo é possível centralizar num único arquivo várias características de vários tipos distintos, de forma organizada, de fácil manutenção e edição. No EPOS, os mais diversos componentes do SO podem ser configurados facilmente através dos traits; veja a seção 4.4 para mais detalhes.

2.5.4 Interface Infladas:

Um conceito da ADESD é a Interface Inflada. Em sistemas orientados à aplicação, famílias de abstrações são frequentemente tratadas como entidades únicas, algo que pode ser vantajoso para o programador da aplicação, já que este não precisaria se preocupar com qual membro em específico desta família ele precisaria usar (Fröhlich, 2001).

Interface inflada basicamente é uma interface que declara os métodos de todas as classes que herdam dela, exportanto assim todos os métodos daquela família de abstrações, como mostra a figura 2.5.4. Deste modo, o desenvolvedor de aplicativo poderia escrever a aplicação inteira em termos da interface inflada, relegando a tarefa de configuração do sistema a um utilitário automatizado. Tal utilitário poderia, através de uma análise sintática do código fonte, escolher quais os membros mais apropriados da família exportada serão associados no momento da compilação (Fröhlich, 2001, p. 56).

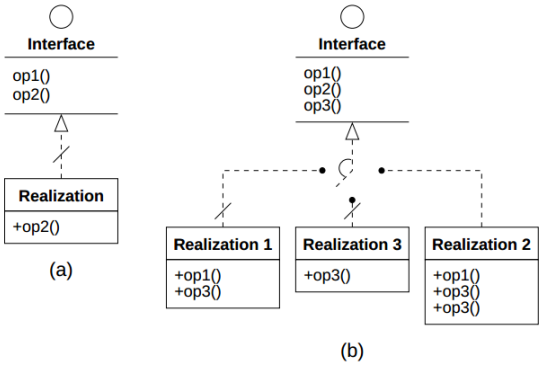

Figura 1 – Exemplos de uso de interfaces infladas (Fröhlich, 2001).

2.6 ARQUITETURA DE MICROPROCESSADORES

Existe uma vasta variedade de sistemas embarcados, de modo que é difícil delinear características que são comuns a todos, portanto nas seções abaixo são colocados alguns conceitos que estão presentes na plataforma escolhida para este trabalho.

2.6.1 Registradores Mapeados em Memória

Uma maneira usada para se comunicar, configurar e ler o estado de um dado componente de hardware é através da leitura e/ou escrita em registradores mapeados em memória. Um registrador mapeado em memória basicamente é uma região fixa da memória, onde uma escrita naquela posição indica uma escrita no registrador lá mapeado. Exemplificando, a UART da Zedboard possui um registrador chamado rcvr_timeout_reg0, responsável por indicar quantos ciclos[3] a UART deve esperar um novo caractere chegar antes de emitir uma interrupção de *timeout*. Este registrador está mapeado na posição de memória 0xE000001C para a UART0. Portanto, para configurar que o número de ciclos esperado seja de 20, basta escrever este número naquela posição. Uma maneira de fazer esta escrita, por exemplo, em C/C++, é: `*((unsigned long*)0xE000001C) = 20;`.

[3]Na realidade é o número de baud_samples que se passaram.

2.6.2 MMU

A MMU (*Memory Managemed Unit*) é um componente responsável por gerenciar a memória de um sistema. É este componente o responsável por traduzir o endereçamento lógico (memória virtual) em endereçamento físico (memória física). Uma das principais vantagens de seu uso é a possibilidade de proteção de memória (entre processos e SO) e facilitação da escrita de aplicativos, pois estes não precisam levar em consideração o mapa de memória do SO.

2.6.3 UART

A UART (Universal Asynchronous Receiver/Transmitter) é um componente que trata da saída e entrada serial do sistema, portanto sendo o componente responsável por transmitir e receber caracteres, sendo necessário para impressão em tela, o que pode ser feito por via USB (usado principalmente para depuração do código). As duas principais funções na classe da UART dentro do EPOS são justamente a put e get, para impressão e leitura de caractere, respectivamente. O Zynq possui duas UARTs.

2.6.4 *Timer*

Timer é um componente de hardware que pode ser configurado para contar um certo número de ciclos. Ao fim da contagem o *timer* emite uma interrupção, que então o processador pode tratar. Este componente é fundamental para um escalonador de processos e *threads*.

2.7 AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento usado para o porte foi o qemu para ARM (`qemu-system-arm`) modificado pela Xilinx, para executar o EPOS num ambiente virtualizado (emulado), pois assim pode-se usar ferramentas como o GDB para a depuração do código de forma rápida. O uso do GDB foi fundamental para o desenvolvimento, já que com ele era possível imprimir o valor de cada registrador em um dado ponto da execução, verificar a memória, executar instruções específicas e etc. Para compilar o EPOS foi usado *cross-compilers* para ARM. As flags usadas para criar um ambiente de depuração com o qemu e o GDB foram as seguintes (com cada comando feito em um

terminal diferente):

```
./qemu-system-arm -kern-dtb ./xilinx_zynq.dtb \
    -no-reboot -nographic -smp 2 -machine xilinx-zynq-a9 \
    -cpu cortex-a9 -kernel img/panda_app -m 512 \
    -fda img/panda_app.img

arm-none-eabi-gdb -ex "target remote :1234"
```

O hardware real também estava disponível, e foi usado quando o qemu não produzia resultados confiáveis. Com o hardware real, para a depuração do código foi usado o JTAG. Para carregar a imagem do SO na placa, foi usada a ferramenta da Xilinx `xmd`, sendo que esta ferramenta trata de fazer o processador apontar para o ponto de entrada da imagem carregada, de tal modo que é possível conectar o GDB através da porta aberta pelo `xmd`, e então executar através do GDB.

3 HARDWARE-ALVO

Zedboard é uma plataforma de desenvolvimento que suporta uma grande variedade de aplicações, visto que ela possui uma boa gama de interfaces e funções para habilitar isto. É dedicada à prototipação e *proof-of-concept*. Em seu interior ela possui um Xilinx Zynq 7000 (Z-7020), que é a arquitetura alvo deste porte. A figura 3 mostra como é a placa fisicamente. Zynq 7020 possui um processador *dual core* Cortex A9, cada core possui sua própria MMU e memória cache L1 (instruções e dados) privada.

O ZYNQ-7000 SOC XC7Z020-CLG484-1 conta com o processador Dual ARM Cortex-A9 MPCore. O Zynq possui 4 graduações de velocidade de *clock*, a comercial (graduação -1), industrial (graduação -1 a -2), estendida (-2 a -3) e expandida (-1), sendo a graduação -1 a menor velocidade, e a -3 a maior (ZYNQ-7000..., 2013).

De acordo com a especificação da Zedboard (ZEDBOARD, 2014), o *clock* máximo do processador é de 667MHZ, portanto, tendo como referência a tabela de dados do Zynq-7000 (ZYNQ-7000..., 2014, p. 13), chegamos à conclusão que a graduação de velocidade do Zynq usado na Zedboard é de -1 (comercial), esta informação se tornará útil mais à frente.

Há disponível 512MB de RAM DDR3, e um SD card de 4GB. A Zedboard suporta conexão com JTAG, saída serial (USB UART) e conexão com a internet (XILINX..., 2014).

A família Zynq 7000 disponibiliza para o desenvolvedor FPGAs, tornando esta plataforma mais configurável e flexível (ZYNQ-7000..., 2013, p. 26). A PS dessa família é a mesma para cada dispositivo onde ela se encontra, entretanto a PL e recursos de entrada e saída variam entre diferentes dispositivos.

3.1 OCM

O *On-chip Memory* é uma pequena memória de 256KB de RAM que fica próxima ao processador, e portanto tem um acesso mais rápido. O OCM pode ser mapeado nos primeiros 256KB do espaço de endereçamento, ou nos últimos 256KB do espaço de endereçamento (ZYNQ-7000..., 2013).

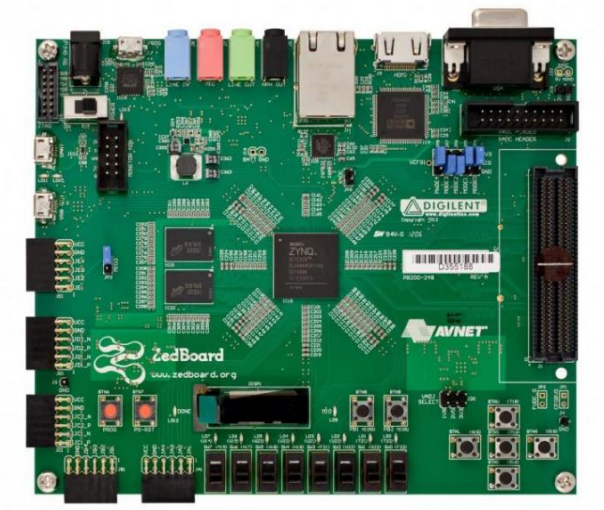

Figura 2 – Zedboard visto de cima.

3.2 MODOS DE OPERAÇÃO

A arquitetura ARMv7 conta com 9 modos de operação (7 por padrão, mais 2 com extensões habilidadas), sendo que o único modo não privilegiado é o modo de usuário, os demais modos padrão possuem o mesmo privilégio dentro do sistema[1]. A principal diferença entre um modo e outro é que cada modo conta com um certo subconjunto privado de registradores banqueados, visiveis somente no modo em vigência, como ilustra a figura 3. A tabela 1 ilustra quais são os modos de operação disponíveis (ARM. . . , 2011, p. 1139). O campo "Codificação" é usado no registrador CPSR para se verificar ou modificar o modo de operação vigente.

Modo Usuário: Modo não-privilegiado de execução. Neste modo somente é possível de se fazer acesso não privilegiado aos recursos do hardware (não podendo acessar as áreas protegidas). Não é possível mudar para outro modo de operação quando neste, a não ser por eventos externos como interrupções.

Modo Sistema: Modo privilegiado de execução. Este modo usa os mesmos registradores que o modo usuário e nenhuma exceção leva a este modo. Somente é possível de entrar nesse modo alterando os bits do regis-

[1] O modo hipervisor possui certas funções que os demais modos não oferecem, mas no que diz respeito à acesso a memória e execução de instruções, ainda é o mesmo nível de privilégio.

Modo do processador	Codificação	Implementado?
User	10000	Sempre
FIQ	10001	Sempre
IRQ	10010	Sempre
Supervisor	10011	Sempre
Monitor	10110	Com extensões de segurança.
Abort	10111	Sempre
Hyp	11010	Com extensões de virtualização.
Undefined	11011	Sempre
System	11111	Sempre

Tabela 1 – Modos do processador. Codificação corresponde aos bits CPSR[4:0].

trador de status do sistema (CPSR); é necessário já estar em algum modo privilegiado para tal operação.

Modo Supervisor: É o modo padrão para no qual processador entra quando uma exceção do tipo *Supervisor Call* é recebida. Para gerar um *Supervisor Call*, usa-se a instrução `svc`. O processador entra neste modo ao se resetar.

Modo Abort: Modo que o processador entra quando recebe uma interrupção do tipo *prefetch abort* ou *data abort*.

Modo Indefinido: Modo que o processador entra quando se tenta executar uma instrução não definida ou mal formada.

Modo FIQ: Modo que o processador entra quando recebe uma interrupção FIQ.

Modo IRQ: Modo que o processador entra quando recebe uma interrupção IRQ.

Modo Hipervisor: Este modo possui alguns privilégios a mais que os demais modos, mas somente existe quando as extensões de virtualização estão ativas (fora do escopo deste trabalho).

Modo Monitor: Modo que o processador entra quando recebe uma interrupção *Secure Monitor Call*, `SMC`. Este modo está fora do escopo do trabalho.

3.3 GIC

O GIC (*Generic Interrupt Controller*) é um componente que centraliza e administra todas as interrupções do sistema, ativando, desativando, masca-

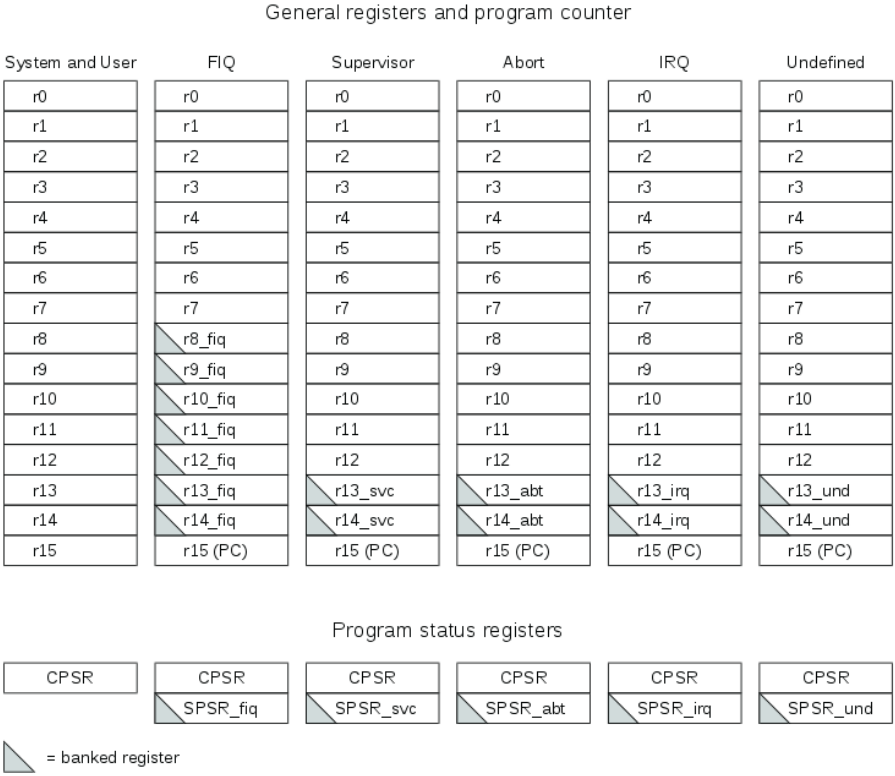

Figura 3 – Registradores banqueados (os modos especiais *hypervisor* e *monitor* não estão contemplados).

rando e priorizando as fontes de interrupção.

3.3.1 Tipos de Interrupção

Interrupção Gerada por Software: Cada CPU pode se interromper, interromper outra CPU, ou ambas CPUs usando SGIs (*Software Generated Interrupts*). Existem 16 interrupções geradas por software, que podem ser geradas escrevendo o número da interrupção ([0-15]), junto com o número da CPU alvo no registrador ICDSGIR. Esta escrita ocorre dentro do barramento privado da própria CPU. Cada CPU possui seu próprio conjunto privado de registradores de SGIs para gerarem uma (ou mais) das 16 SGIs possíveis

(ZYNQ-7000..., 2014, p. 216). É possível limpar uma interrupção lendo o registrador ICCIAR (*Interrupt Acknowledge*) ou escrevendo 1 nos bits correspondentes do registrador ICDICPR (*Interrupt Clear-Pending*).

Interrupções de Periféricos Privados da CPU:

Interrupções de Periféricos Compartilhados: Existem cerca de 60 interrupções de diversos modulos que podem ser roteadas para um ou ambos processadores, ou para a lógica programável. O GIC é responsável por administrar estas interrupções. Além das interrupções IRQ, há também interrupções do tipo FIQ e geradas por software.

FIQ *Fast Interrupt*: O modo FIQ possui um número maior de registradores banqueados que os demais modos (R8 até R13, além do SP, LR e SPSR), fazendo com que não seja necessário fazer troca de contexto. FIQs também tem mais alta prioridade que qualquer IRQ, pontanto este tipo de interrupção é apropriado para aplicações de tempo real de usuário único. Se multiplas aplicações tempo real usarem FIQ, pode haver conflitos de interesse que podem fazer um processo perder um *deadline* (ARM..., 2011, p. 66).

Interrupção de Software: Existe uma instrução no assembly do ARMv7 que permite que seja feita uma interrupção de software (`swi`). Esta instrução pode ser acompanhada de um número de 8 bits. Esta interrupção leva ao modo supervisor.

A Zedboard possui também as seguintes exceções[2]:

Instrução não definida (*Undefined Instruction*): Há duas situações que geram esta interrupção: Quando se executa instruções de coprocessador que não são reconhecidas por este, ou quando se executa instruções que não possuem significado para o processador (ARM..., 2011, p. 36). Normalmente este tipo de interrupção acontecerá quando o processador estiver lendo e tentando executar lixo de memória.

Prefetch Abort: Interrupção que leva ao modo *Abort*. Este tipo de interrupção é sinalizado pelo sistema de memória (MMU). Através da MMU, é possível de se marcar certas regiões de memória como não executáveis; isto é importante de se fazer em regiões de memória que possuem dados sensíveis, de modo que quando o processador tentar ler aquela área para executar num futuro próximo, a MMU envia um sinal invalidando aquilo que foi lido. Quando o processador tenta executar uma instrução que tenha sido anteriormente invalidada, uma interrupção do tipo *prefetch* ocorre (ARM..., 2011, p. 58). Note que é possível do processador ler aquela região para executar em seguida, mas ainda não gerar esta interrupção; isto ocorre quando o fluxo é desviado antes da tentativa de executar aquela instrução (por um *branch* por

[2]Em conceitos básicos está colocada a diferença entre interrupções e exceções

exemplo).

Data Abort: Assim como a *prefetch abort*, esta interrupção também leva ao modo *Abort* (e somente estas duas interrupções). Este tipo de interrupção pode acontecer quando uma instrução tenta acessar uma região de memória que o modo atual de execução não tenha permissão para acessar. Acesso a regiões de memória virtual não mapeadas pela MMU também geram esta interrupção.

3.4 *TIMERS*

Cada um dos *cores* possui um *timer* privado de 32 bits e ambos *cores* compartilham um *timer* global de 64 bits. Estes *timers* trabalham numa frequência sempre igual à metade da frequência da CPU. No nível de sistema (*system-level (PS)*), há dois grupos independentes de *timers*, cada grupo com 3 *timers*.

3.5 *CLOCKS*

O *clock* principal do sistema, chamado aqui de PS_CLK (*Processing System Clock*), é responsável por alimentar as 3 PLLs[3] do sistema, sendo cada uma dessas PLLs responsável por uma parte diferente do sistema (ZYNQ-7000. . . , 2013, p. 622). O PS_CLK é um *clock* de baixa frequência, ficando entre 30 a 60 MHz (PS_CLK é igual a 33.33 MHz no caso da Zedboard (ZEDBOARD. . . , 2014, p. 19)), sendo este multiplicado por cada uma das 3 PLLs para que o sistema funcione com velocidades maiores[4]. As 3 PLLs são:

- **I/O PLL:** Responsável por produzir o sinal de *clock* para os dispositivos de entrada e saída.
- **DDR PLL:** Responsável por produzir o sinal de *clock* para as memórias da plataforma.
- **ARM PLL:** Responsável por produzir o sinal de *clock* do restante do sistema, incluindo os processadores.

A FPGA da Zedboard possui um *clock* próprio e exclusivo. A figura 4 ilustra como estão dispostos estes *clocks*.

[3] *Phase-Locked Loop*. É um sistema de controle que gera uma saída cuja fase é relacionada à fase do sinal de entrada. Pode ser usada para estabilizar um sinal e também multiplica-lo.

[4] Este clock pode ser multiplicado por um número de 1 a 127, e é multiplicado por 26 por padrão.

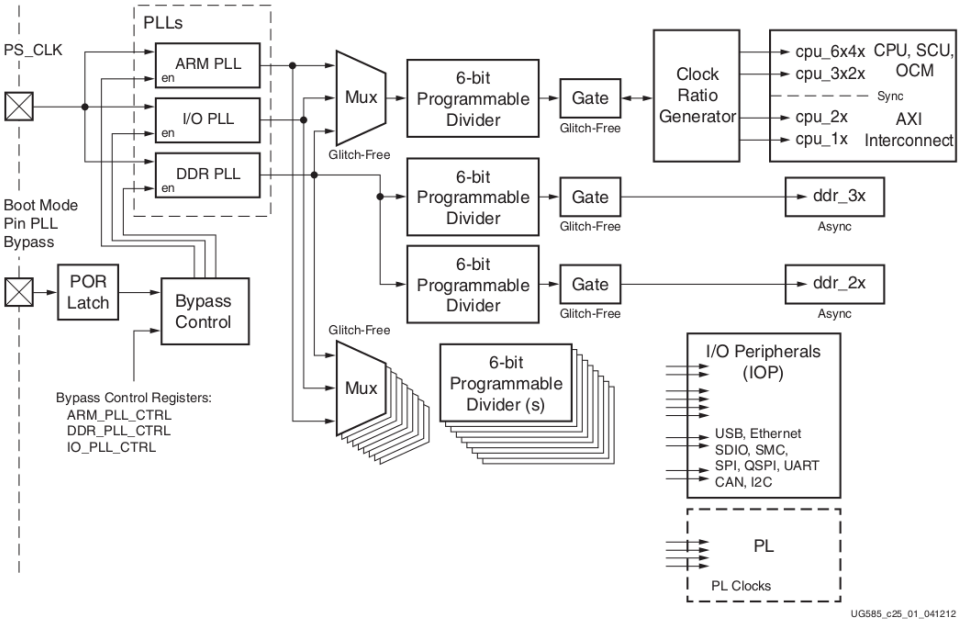

Figura 4 – Diagrama de blocos dos *clocks* disponíveis no Zynq. Note os *clocks* da tabela 2 no canto superior direito da imagem.

A Zedboard pode operar em dois modos (ou velocidades), denominados pela pela razão 6:3:2:1 e 4:2:2:1, abreviados como 6:2:1 e 4:2:1. Para alternar entre estes dois modos de velocidade, basta escrever 1 ou 0 no registrador CLK_621_TRUE. Estes números indicam quantas vezes cada *clock* multiplica o *clock* de base CPU_1x, sendo este CPU_1x um *clock* derivado da ARM PLL, dividido por algum fator (configurável).

Há 4 *clocks* independentes, chamados de CPU_6x4x, CPU_3x2x, CPU_2x e CPU_1x. Esta nomenclatura dos *clocks* indica o fator pelo qual o CPU_1x é multiplicado em cada modo. O primeiro número do nome indica o fator multiplicativo daquele *clock* no modo 6:2:1, e o segundo número indica o fator multiplicativo no modo 4:2:1. Por exemplo, no modo **6**:2:1, o CPU_**6x**4x multiplica o CPU_1x 6 vezes, e no modo **4**:2:1, o CPU_6x**4x** multiplica o CPU_1x 4 vezes. A tabela 2 ilustra a velocidade de cada *clock* em cada um dos dois modos.

Clock	*Clock Ratio Mode*	Máxima frequência da CPU
CPU_6x4x	6:3:2:1	667 MHZ
CPU_3x2x		333 MHZ
CPU_2x		222 MHZ
CPU_1x		111 MHZ
CPU_6x4x	4:2:2:1	533 MHZ
CPU_3x2x		267 MHZ
CPU_2x		267 MHZ
CPU_1x		133 MHZ

Tabela 2 – Máximas frequências possíveis para cada configuração de *clock*. Para uma lista mais completa (com as diferentes graduações de *clock*), veja (ZYNQ-7000. . . , 2014, p. 13).

4 EPOS

EPOS (Embedded Parallel Operating System) é um sistema operacional orientado a aplicação, cujo design chama-se ADESD (*Application-Driven Embedded System Design Method*), proposto por Fröhlich (Fröhlich, 2001). A ideia central do EPOS é prover um sistema operacional mínimo, de modo a minimizar o *overhead* da existência de um sistema operacional, deixando o processador livre para executar a aplicação do desenvolvedor (EPOS..., 2014).

Como o objetivo é criar um ambiente em que o desenvolvedor possa rapidamente produzir suas aplicações, EPOS provê vários utilitários comumente usados em aplicações, como filas, listas, tabelas de hashs, vetores, semáforos, OStream (para imprimir na tela), números aleatórios, cálculo de CRC e etc. Além destes utilitários, EPOS também provê uma série de componentes como threads, alarmes, cronometros, *heaps* e meios para acessar a rede (internet).

4.1 ARQUITETURA DO EPOS

Linguagem e paradigma: O EPOS é escrito em C++, e não em C, como é tradicionalmente feito. Como paradigma, é usado orientação à objetos, então cada componente do EPOS está encapsulado por uma classe (como a heap, timer, thread, etc). No EPOS também é muito usado conceitos como herança e metaprogramação estática (que será abordado mais a frente).

Mediadores de Hardware: Dentro da arquitetura do EPOS há o conceito de mediadores de hardware, que são os componentes (ou classes) dependentes de plataforma. Idealmente, as únicas classes que precisam ser modificadas e/ou reimplementadas são os mediadores. Há mediadores específicos da placa, como para a Pandaboard, a Zedboard e etc; abstraídos sob o nome de *machine* e mediadores específicos de um processador, abstraídos sob o nome de *architecture*. No código do EPOS, estes mediadores encontram-se nas pastas *mach* e *arch*.

Mediadores de hardware são uma alternativa ao tradicional uso de VMs[1] e de HAL[2], proposta por Fröhlich em seu trabalho *Application-Oriented System Design* (Fröhlich, 2001). O problema do uso de VMs é o seu *overhead* causado devido à tradução das operações da VM em código nativo. Já o uso de um HAL incorre no problema da manutenibilidade e dificuldade de adap-

[1] Virtual Machines.

[2] Hardware Abstraction Layer

ção à novas arquiteturas muito distintas entre si (Polpeta; Fröhlich, 2004). O HAL não conseguiu passar pela "prova do tempo", e já está sendo considerado obsoleto por distribuições GNU/Linux populares, como o Ubuntu (LINUX..., 2010), sendo chamado de "uma grande não-manutenível bagunça monolítica" (UBUNTU..., 2014).

4.2 MODOS DE COMPILAÇÃO

Para compilar o EPOS, usa-se a ferramenta `make`, através de um conjunto de `makefiles`. As configurações referentes à compilação podem ser encontradas no arquivo `makedefs`, onde pode-se mudar o *cross compiler* a ser usado, o modo de compilação (que será descrito a seguir), arquitetura desejada e etc.

O EPOS pode ser compilado de três formas diferentes:

Library O sistema é compilado junto com a aplicação, sem distinção de espaço de endereçamento, existência de modo usuário nem chamadas de sistema.

Builtin Semelhante ao *library*, com a diferença de que o SO é colocado nos endereços mais altos, e a aplicação nos endereços mais baixos (ambos ainda usam o mesmo espaço de endereçamento), em contraste ao *library* que mistura os dois. O principal uso desse modo é para depuração e desenvolvimento (EPOS..., 2013).

Kernel Existe dois espaços de endereçamento diferentes, com modo usuário e kernel, com uma interface de chamada de sistema entre eles. Existe o conceito de *task*.

Durante o processo de compilação do EPOS, duas ferramentas próprias são chamadas: `eposcc` e `eposmkbi`. O `eposcc` é um *script* auxiliar do processo de compilação, é nele em que é diferenciado o modo de compilação do EPOS, e é nele que é acertada a ordem de ligação (*linkage*) dos construtores globais do EPOS, algo muito importante para a inicialização do SO, como é descrito na seção 4.3. Já o `eposmkbi` é uma ferramenta cujo objetivo é criar uma imagem "bootável" do SO com a aplicação. Esta imagem é uma imagem de virtualização de disco, com as metainformações necessárias já adicionadas a ela (FISCHER, 2013). Estas metainformações são adicionadas também à `struct System_info`, para que seja posteriormente usada pelo EPOS durante sua inicialização.

4.3 INICIALIZAÇÃO

O EPOS inicia executando um conjunto de códigos escritos em assembly, chamados de crts. No crt, a primeira tarefa feita na inicialização do EPOS é a configuração das pilhas do sistema. Feito isto, o EPOS trata de limpar a seção .bss, para futuro reuso. Antes de ser chamado o construtor dos demais objetos globais, a UART e Display são inicializados, assim é possível mais facilmente depurar o código dos construtores. Então o EPOS começa a chamar o construtor global de cada componente do sistema. O crt consegue localizar onde está localizado cada construtor devido à forma como o EPOS foi compilado e ligado (*linked*). A figura 5 ilustra num diagrama de sequência a ordem de construção dos objetos globais.

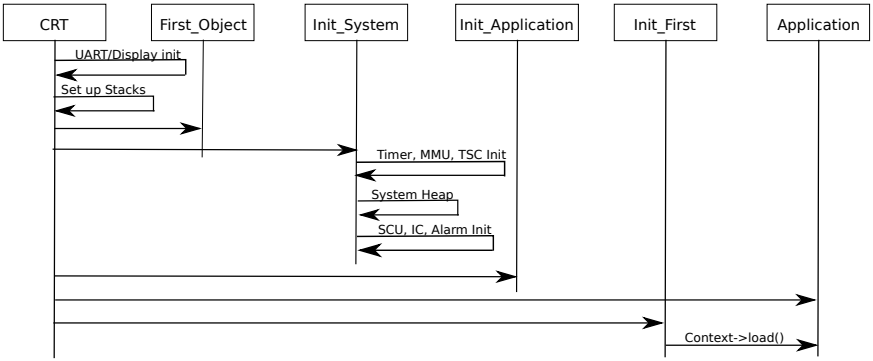

Figura 5 – Diagrama de sequência da inicialização do EPOS

Primeiro é construído o `First_Object`, cuja principal função é apenas ser um ponto de entrada conhecido para o primeiro objeto do sistema. Em seguida, é construído `Init_System`. É neste construtor que todos os mediadores de hardware são construídos, com exceção da UART que já foi previamente construída no crt. Neste construtor, primeiramente é inicializado o *timer*, MMU, e, se disponível, o TSC (*Time Stamp Counter*). Em seguida o construtor aloca uma porção de memória para futuro uso da heap do sistema. Feito isto, o SCU (*Snoop Control Unit*), tratador de interrupções e alarme são também construídos.

`Init_Application` é responsável por alocar a heap da aplicação, através da requisição de páginas da MMU. `Init_First` chama o inicializador das *threads*, que então inicializa a *thread* da aplicação e a *idle thread*, que é uma *thread* especial que é executada quando o escalonador não encontra outra *thread* pronta para executar. Em seguida à criação destas *threads*,

é chamado `context->load()` da *thread* da aplicação, fazendo com que o fluxo de execução seja finalmente transferido para a aplicação.

4.4 TRAITS

Existem 4 arquivos `traits.h` que devem ser levados em consideração em um porte, dois deles devem ser completamente reescritos. Os arquivos `./include/traits.h` e `/include/system/traits.h` (onde '.' é a pasta raiz do código) possuem configurações gerais do EPOS, que, a princípio, devem ser independentes de arquitetura. Na prática há alguns pequenos ajustes que devem ser feitos nesses arquivos, pois é lá que se define, por exemplo, se o EPOS trabalhará em um processador *multicore*, se utilizará *scratchpad*, quais componentes estarão em modo de depuração e etc; entretanto isto se resume a trocar o valor de algumas variáveis de *true* para *false* e vice-versa.

Os outros dois arquivos são `./include/mach/zynq/traits.h` e `./include/arch/armv7/traits.h`. Essa divisão é necessária pois, como dito anteriormente, é possível de um mesmo processador estar em diferentes *machines*, e, caso seja necessário fazer um porte para aquela plataforma, bastaria modificar os arquivos da pasta *mach*, deixando os da pasta *arch* praticamente intactos, o que facilita novos portes.

O arquivo `./include/arch/armv7/traits.h` trata das opções específicas do processador, portanto é lá que opções como *endianess*, velocidade de *clock*, número de *cores*, tamanho da *heap* e *stacks*, bem como outras opções pertinentes ao mapeamento de memória e opções da MMU podem ser configuradas.

No arquivo de `traits` da *machine*, ficam as opções de configuração de componentes como a UART, controlador de interrupções, *timer*, e qualquer componente de interfaceamento externo à placa (rede, por exemplo). Componentes podem ser facilmente ativados ou desativados nestas opções.

4.5 INTERRUPÇÕES

Como o EPOS foca no desempenho da aplicação, em seu design opta-se por utilizar a menor quantidade possível de interrupções, e evitá-las sempre que possível, isto contribui também para que o sistema seja mais previsível. Se a aplicação não precisar de componentes específicos que necessitem interrupções (como um rádio, por exemplo), a única interrupção a ser tratada será a do *timer*, já que o escalonador e alarm necessitam dele.

A sequência de chamadas de função, dado o recebimento de uma in-

terrupção de *timer* está ilustrada na figura 6. Exception é um código em assembly cujo objetivo é salvar o contexto no recebimento da chamada, chamar o tratador de interrupções padrão, e restaurar o contexto após isto. O tratador de interrupções do *interrupt controller* (IC) possui um vetor de tratadores de interrupção, onde para cada número de interrupção, há uma entrada correspondente no vetor. Este tratador então lê o registrador responsável por armazenar o número da interrupção gerada (esta parte é dependente de arquitetura, na Zedboard uma interrupção de timer tem número 29, por exemplo), indexa seu vetor e chama o tratador apropriado. Chegando no tratador do *timer*, este chama os tratadores de todos os componentes que dependem de um *timer*, neste caso o `Alarm` e `Scheduler`.

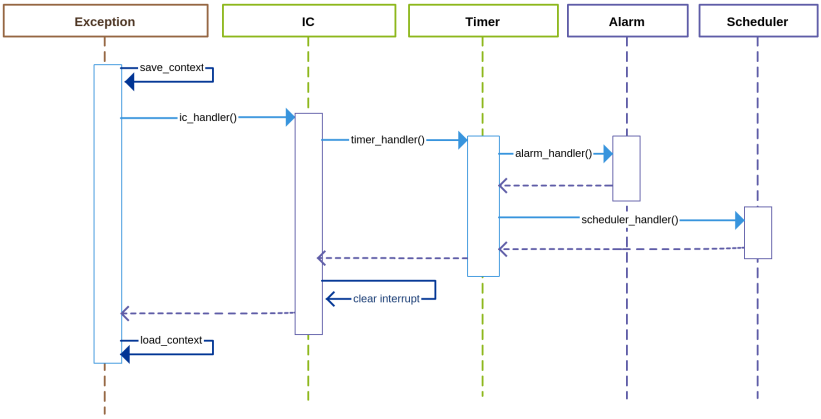

Figura 6 – Sequência de chamada de tratadores de interrupção. A interrupção gerada neste exemplo foi uma de *timer*.

.

4.6 GERENCIAMENTO DE MEMÓRIA

Uma das principais funções de um sistema operacional é o gerenciamento de memória. Isto inclui organizar a memória (mapeamento de memória, definição do que cada região da memória guarda, etc) e gerenciamento (manter registro de páginas livres, definir permissões, etc).

Para entendermos como isto é feito, primeiro é necessário entendermos como funciona a abstração da MMU no EPOS. Isto se dá com 4 classes principais, a `ARMv7_MMU`, que possui a lista de memórias livres, a `Page_Table`,

que mapeia porções de até 1MB de endereço virtual para físico, `Chunk`, que nada mais é que um conjunto de `Page_Tables`, e, finalmente, `Directory`, que é um conjunto de `Chunks` (ou `Page_Tables`).

A classe `ARMv7_MMU` possui uma uma lista de porções de memórias livres, gerenciadas pelas funções `alloc` e `free`. A granularidade destas porções de memória é a de *frames* [3]. Note que esta lista utiliza os endereços **físicos** destas porções de memória.

A tabela de páginas da MMU do ARMv7, assim como a do IA32, possui dois níveis. Na nomenclatura da ARM, a tabela de nível mais alto é a tabela L1 (de *level*), onde cada entrada desta tabela aponta para outra tabela (uma tabela L2). Cada entrada da tabela L2 possui o endereço físico correspondente ao endereço virtual requisistado. A maneira como esta tabela é usada para traduzir um endereço virtual para um físico varia de arquitetura para arquitetura, sendo esta parte explicada detalhadamente na seção 5.5.

Nas abstrações do EPOS, a classe que gerencia a tabela L2 é a `Page_Table`, e a classe que gerencia a tabela L1 é a `Directory`. Portanto uma `Page_Table` é apenas um vetor, onde em cada posição está escrita uma posição de memória física, junto de algumas *flags* (que definem permissões, "cacheabilidade" e etc). O número máximo de entradas numa `Page_Table` é dependente de arquitetura. No ARMv7 são 256 entradas de 4 bytes (mapeando até 1MB de memória portanto), e no IA32 são 1024 entradas (que mapeiam até 4MB), cada página (*frame*) possuindo 4 KB.

`Chunk` é uma classe bastante importante, e, apesar de simples, é importante que se tenha um bom entendimento sobre como ela funciona. Cria-se um `chunk` para alocar uma certa porção de memória, em um espaço de endereçamento próprio.

Esta classe recebe dois parâmetros: A quantidade de bytes a serem alocados, e as *flags* a serem usadas. `Chunk` então calcula quantas tabelas de páginas serão necessárias para mapear a quantidade de bytes requisitada, e então cria estas páginas contiguamente no espaço de endereçamento virtual (em qualquer lugar da memória, a ser decidido pela função `alloc`).

`Chunk` então chama a função `map` de `Page_Table` para mapear aquela porção da memória. Cada *frame* é requisitado pela função `alloc`, o que significa que cada `Page_Table` pode estar apontando para qualquer posição da memória, não estando os *frames* necessariamente contiguos na memória física, apesar que eles serão **contíguos** no espaço de memória virtual mapeada num mesmo `Chunk`.

Portanto esta classe efetivamente aloca uma porção de memória em

[3] Frame é uma porção de tamanho fixo de memória física (normalmente 4KB). A diferença entre uma página e um frame é que um frame refere-se à uma porção da memória física, enquanto uma página refere-se à uma porção de memória virtual.

um espaço de endereçamento próprio, este fato será bastante importante quando discutirmos a criação das *heaps* de usuário e do sistema.

Já a classe directory é responsável por gerenciar e unificar todas estas porções de memória. Sua principal função é a `attach`, que tomando como parâmetro um `Chunk`, mapeia cada uma de suas páginas contiguamente (isto é, cria uma entrada em `directory` que aponta para uma página criada em `Chunk`, para cada página dele).

A figura 7 ilustra a interação entre os conteúdos das entradas de `Directory` e `Page_Table` com a memória física. A quantidade de bits usada para indexar as tabelas é dependente de arquitetura. Aqui, os primeiros 12 bits indexam a tabela de `Directory`, os 8 bits seguintes indexam a tabela de `Page_Table`, e os últimos 12 bits são simplesmente copiados do endereço virtual para o físico. Portanto, no exemplo, do endereço 0xAB2FF123, o número 0xAB2 será usado para indexar a tabela de nível 1 (`Directory`), o número 0xFF indexará a tabela de segundo nível (`Page_Table`), onde estará escrito o endereço físico correspondente àquele endereço virtual. Mais detalhes sobre este processo na seção 5.5, em particular na imagem 14.

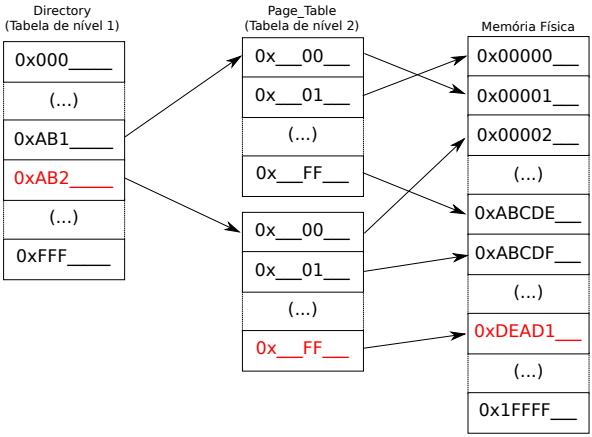

Figura 7 – Interação entre Directory e Page_Table com a memória física. Neste exemplo, o endereço virtual 0xAB2FF123 seria traduzido para o endereço físico 0xDEAD1123. A última posição de memória física indexável neste exemplo é 0x1FFFF___ pois a Zedboard dispõe de 512MB.

`Directory` não sabe a qual `Chunk` uma determinada `Page_Table` pertence, e a tradução de endereços pode ocorrer independente desta informação. Na figura 8 é ilustrada a interação entre as classes discutidas.

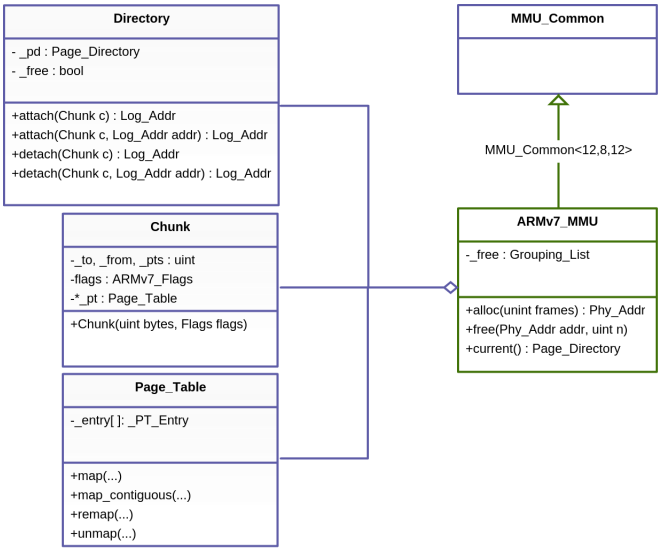

Figura 8 – Diagrama de classes da MMU.

4.6.1 Heaps

A *heap* é responsável por armazenar e gerenciar os dados dinâmicos criados durante a execução da aplicação e do SO (a cada `malloc/new` executado, por exemplo). Existem duas heaps: A do sistema, e a do usuário. É possível compilar o EPOS com apenas a *heap* do sistema, quando o desenvolvedor decide que a aplicação deverá rodar em modo privilegiado, não sendo usado portanto o modo usuário (isto é configurável nos `traits`).

A memória da *heap* é alocada através de um `Chunk`. É neste ponto que pode-se atribuir *flags* que limitam as permições de acesso (a *heap* de usuário é alocada sem restrições de acesso, já a do sistema aloca uma porção de memória que só pode ser acessada em modo privilegiado). É por usar um `Chunk` também que do ponto de vista da aplicação, é como se ela tivesse acesso à todo o espaço de memória. Note que do ponto de vista da MMU (e sua lista de memórias livres), é como se a porção de memória da heap estivesse em uso, já a heap vê sua porção de memória como livre.

A heap trabalha com uma lista de memórias livres (igual à da MMU), só que ao contrário da MMU, que aloca somente páginas (frames) inteiros de memória, a heap aloca quantidades de memória com a granularidade de

bytes.

Para que a *heap* do sistema e a *heap* do usuário seja corretamente usada, faz-se no EPOS *overload* do operador `new` e `delete`, e *placement new*. Por exemplo, um `new A()` chamaria as seguintes funções:

```
inline void * operator new(size_t bytes) {
    return malloc(bytes);
}
inline void * malloc(size_t bytes) {
    if(Traits<System>::multiheap)
        return Application::_heap->alloc(bytes);
    else
        return System::_heap->alloc(bytes);
}
```

O compilador se encarrega de avaliar o `sizeof` de `A`, e envia como parâmetro para o operador `new`. `multiheap` é um atributo dos `traits` que define se o sistema possui uma *heap* de usuário separada da *heap* do sistema ou se é usada somente uma única *heap*.

Para o sistema alocar memória usando sua própria *heap* com o operador `new`, faz-se um *overload* do *placement new*, como segue:

```
enum System_Allocator {SYSTEM};

inline void * operator new(size_t bytes, const EPOS::
    System_Allocator & allocator) {
    return EPOS::System::_heap->alloc(bytes);
}
```

Feito isto, para alocar, dentro do EPOS, alguma porção de memória utilizando o operador `new` com a *heap* do sistema, basta escrever `(SYSTEM)` logo após o `new`, por exemplo:

```
_timer = new (SYSTEM) Alarm_Timer(handler);
```

4.7 ESCALONADORES

O EPOS suporta diversos escalonadores, desde os de propósito geral até escalonadores de tempo real *multicore*. Do primeiro tipo, temos o *Round Robin* e *First-Come, First-Served*, do segundo, *Rate Monotonic*, *Deadline Monotonic* e *Earliest Deadline First* (global, particionado e agrupado), e também o escalonador descrito em (Gracioli, 2014).

O EPOS é projetado de modo a permitir grande reuso ao se escrever um escalonador. Por exemplo, há três escalonadores que utilizam a política EDF (*Earliest Deadline First*), que nada mais é que um critério de escolha de *thread*, onde neste caso procura-se escolher aquela cujo *dealine* está mais

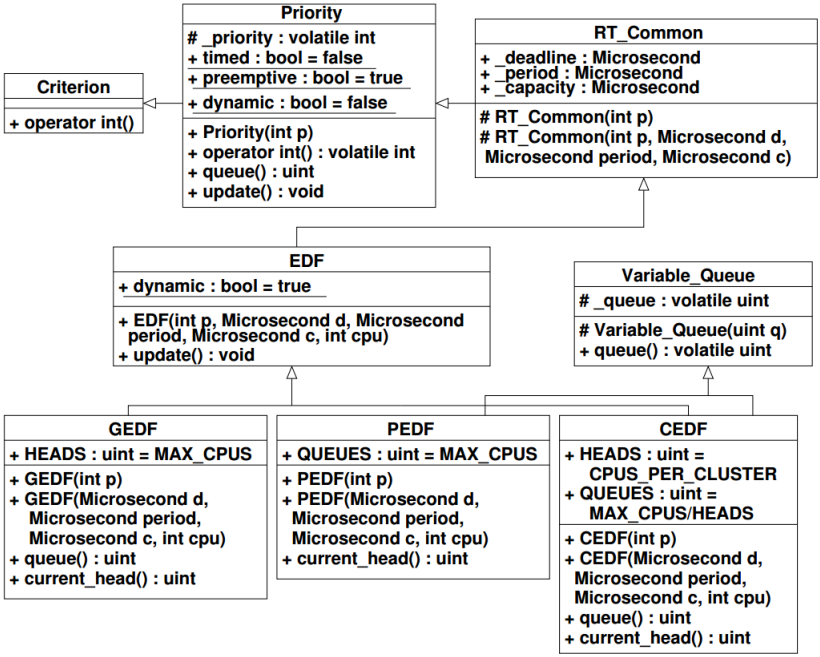

Figura 9 – Diagrama de classes para os escalonadores que usam a política *Earliest Dealine First* (Gracioli, 2014, p. 184).

próximo, por isto separou-se esta parte das especificidades que existem entre os escalonadores.

A figura 9 ilustra a relação entre as classes dos escalonadores que herdam de EDF. Não é o objetivo deste trabalho entrar em detalhe nas especificidades dos diferentes escalonadores do EPOS, entretanto será colocada uma breve descrição destes três escalonadores EDF para que a figura fique clara.

PEDF: O *partitioned* EDF (particionado) possui uma fila de tarefas a serem escalonadas para cada processador. A escolha de que fila cada tarefa é atribuida é feita de forma estática. Cada tarefa executa sempre no mesmo processador, não havendo migração destas.

GEDF: O EDF global utiliza uma única fila de tarefas, e atribui uma tarefa toda vez que um processador fica livre. A vantagem deste escalonador sobre o PEDF é que ele consegue escalonar conjuntos de tarefas que o PEDF não conseguiria, entretanto seu desempenho costuma ser menor

e menos previsível, pois migrar uma tarefa de um processador para o outro é custoso, e gera um mal aproveitamento das memórias *cache*.

CEDF: Este é o caso geral do PEDF e GEDF. Neste escalonador há uma fila para cada grupo de processadores (*cluster*). Por exemplo, numa arquitetura que usa 8 processadores, poder-se-ia fazer 4 grupos de 2 processadores cada, totalizando 4 filas. Somente há migração de tarefas entre processadores de um mesmo grupo. Caso o grupo seja do tamanho do número de cores, tem-se o GEDF, e se cada grupo tiver apenas um processador, obtem-se o PEDF.

4.8 RELAÇÃO MEDIADORES-ABSTRAÇÕES

Esta seção discute a relação entre os mediadores de hardware do EPOS (dependentes de arquitetura) com as abstrações do EPOS que usam estes mediadores (independente de arquitetura). O objetivo aqui é mostrar que estas abstrações não precisam ser alteradas entre as diferentes arquiteturas que o EPOS suporta, estando todas as diferenças arquiteturais isoladas em seus mediadores.

Cada mediador de hardware (que chamaremos apenas de "mediador" deste ponto em diante) possui uma classe genérica que o representa. Esta classe é a mesma para todas as arquiteturas, e cada novo mediador implementado deve estender esta classe, deste modo pode-se manter a uniformidade das interfaces de cada novo mediador. Por exemplo, no caso da UART, cada novo mediador da UART precisa estender a classe `UART_Common`, como ilustra a figura 10. Note que o EPOS possui suporte para diversas arquiteturas, no diagrama são colocadas apenas duas por simplicidade. A figura 8 da seção 4.6 também ilustra isto.

A resolução do mediador da-se em tempo de compilação, através de metaprogramação estática e do uso de scripts automatizados. Para escolher a arquitetura alvo para compilar o EPOS, basta alterar o arquivo de traits `./include/system/traits.h`, trocando os valores das variáveis `ARCH` e `MACH`. Durante o processo de compilação, usando a ferramenta `sed`, o arquivo `./include/system/config.h` é alterado para alterar as variáveis `ARCH` e `MACH` para as escolhidas pelo usuário. Feito isto, macros são usadas para a resolução da inclusão dos *headers* corretos para a arquitetura escolhida. Segue abaixo algumas destas macros:

```
#define __HEADER_ARCH(X)          <arch/ARCH/X.h>
#define __HEADER_MACH(X)          <mach/MACH/X.h>
```

Para que o código independente de arquitetura do EPOS possa se re-

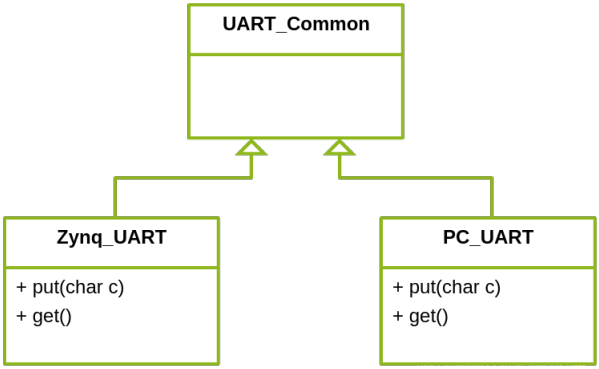

Figura 10 – Diagrama de classes da UART.

ferenciar às classes dos mediadores através de um nome genérico, para cada porte é feito um arquivo `config.h` onde são definidos tipos com nome padronizado que se referem aos mediadores daquela arquitetura. Por exemplo, um trecho do `config.h` do Zynq segue abaixo:

```
typedef ARMV7      CPU;
typedef ARMV7_MMU  MMU;
typedef Zynq       Machine;
typedef Zynq_IC    IC;
typedef Zynq_UART  UART;
typedef Zynq_Timer Timer;
```

Com isto, todos os mediadores ficam abstraídos do ponto de vista do restante do sistema operacional, e podem ser trocados através da simples alteração de duas variáveis no arquivo de `traits`, seguida de nova compilação.

5 IMPLEMENTAÇÃO DOS MEDIADORES DE HARDWARE

Nesta seção será discutido como foi o porte de cada mediador de hardware, explicando as decisões tomadas.

O processo de inicialização do sistema envolve as seguintes ações (ZYNQ-7000..., 2014, p. 110):

1. Definir a tabela de vetores.
2. Invalidar *caches*, TLB, *branch predictor*
3. Preparar tabelas de tradução de página.
4. Configurar as pilhas dos diferentes modos de execução.
5. Carregar o endereço base da tabela de páginas no registrador apropriado para ser usado pela MMU.
6. Ativar a MMU.
7. Ativar a L2, ativar a L1.
8. Pular para o ponto de entrada a aplicação.

Estes e outros passos (como inicialização do GIC, timer, UART, CPU, etc) são explicados em detalhe nas seções que seguem. O primeiro passo a ser feito para iniciar um novo porte é incluir em `./include/system/types.h` os nomes das classes dos mediadores que serão implementados e criar as pastas necessárias para colocar estas classes, no caso deste porte, foram criadas as pastas `./include/mach/zynq/`, `./src/mach/zynq/`, `./include/arch/armv7/` e `./src/arch/armv7/`.

5.1 MEDIADOR DA UART

A inicialização da UART foi feita de acordo com o sugerido pelo manual em (ZYNQ-7000..., 2013, p. 554). Nesta seção será comentado as decisões tomadas na configuração inicial da UART, em particular por causa dos momentos em que o manual exigia que o desenvolvedor tomasse uma decisão.

A primeira decisão que foi necessária é a de escabelecer qual será a taxa de transmissão (*Baud Rate*) da UART.

Primeiramente foi necessário configurar o *clock* de referência da UART. Para isto, deve-se dividir o *clock* que vem do I/O PLL (que por sua vez deriva

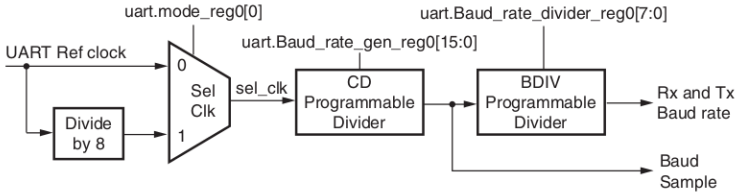

Figura 11 – Esquemático de como é criada a taxa de transmissão.

do PS_CLK, que é o *clock* geral do sistema). Recomenda-se dividir o *clock* da I/O PLL de modo a se obter 50 ou 33 MHz. O *clock* da I/O PLL, por padrão, multiplica o PS_CLK (de 33.33 MHz) por 26, resultando num *clock* de 866 MHz. No manual diversas vezes é usado como exemplo para o *clock* de referência (que é mostrado sob o nome de `UART Ref clock` na figura 11) da UART como 50 MHz, portanto, arbitrariamente, escolheu-se esse valor. Logo, deve-se configurar o registrador UART_CLK_CTRL, responsável por configurar o *clock* de entrada da UART, para dividir este *clock* vindo da I/O PLL por 17 ($866/17 = 50$).

Agora, com este *clock* estabelecido, que vamos chamar de sel_clk (de acordo com a nomenclatura do manual), devemos calcular quanto será a taxa de transmissão. Após alguma pesquisa em fóruns de desenvolvedores de software básico, foi observado que uma taxa de transmissão de 9600 bps é bastante comum, portanto, assumindo este valor, a próxima etapa é configurar os dois divisores de *clock* que ajustam a taxa de transmissão, como indicado na figura 11.

O primeiro divisor chama-se CD (*clock divider*), que configura a constante para se dividir o *clock*, e BDIV um segundo divisor usado para sobreamostragem, organizado como mostrado na figura 11. O valor da taxa de transmissão final é calculado da seguinte forma:

$$\text{taxa de transmissão} = \frac{sel\_clk}{CD \times (BDIV + 1)} \tag{5.1}$$

O valor padrão de BDIV é 15, portanto, fixando-se este valor e resolvendo a equação por CD, temos que $CD = 325$. Configurando-se estes valores nos seus respectivos registradores, obtém-se a taxa de transmissão desejada de 9600 bps.

Após estas configurações, dentre outras que o manual descreve, a UART está pronta para ser usada. Os dois principais métodos usados da UART são o `put` e o `get`, o primeiro escreve um caractere na saída serial,

sendo que este pode ser lido, por exemplo através de uma entrada USB, para imprimir estes caracteres numa tela; algo muito útil para depuração.

5.2 MEDIADOR DO *TIMER*

Um *timer* permite contar um certo número de ciclos, e, ao final da contagem, ele emite uma interrupção ao processador, para que então o processador trate este evento. Entretanto note que é possível de existir mais *timers* sendo usados logicamente do que *timers* físicos disponíveis, significando que um mesmo *timer* deve conseguir servir a mais de uma requisição simultaneamente.

Portanto não podemos apenas configurar um *timer* para contar initerruptamente até passar o tempo que desejamos, do contrário novas requisições sobreescreveriam a anterior. Para ilustrar, suponha que se queira contar por 10 segundos, como o *clock* do *timer* é de 333 MHz (periodo $\frac{1}{333\times10^6}$s), bastaria configurar o *timer* para contar por $10 \times 333 \times 10^6$ ciclos e então chamar o *handler* associado à interrupção gerada quando o *timer* chegar em zero.

Agora imagine que, no cenário acima, enquanto o *timer* ainda está servido àquela solicitação de contagem, apareça outra solicitação, de um alarme por exemplo, e queria contar por 20 segundos. Se esta solicitação sobrescrever o registrador de configuração do *timer*, a solicitação anterior não terá seu pedido atendido a tempo. Note também que o escalonador de processos também estará usando este *timer*.

Para resolver este problema, na arquitetura do EPOS existe o conceito de ticks (algo parecido com o que se faz no Linux), onde se configura um *timer* para gerar interrupções em um intervalo regular, intervalo este que deve ser pequeno o suficiente para poder atender a demanda de contagens de pequenos valores, assim como não ser pequeno demais ao ponto de gastar mais processamento tratando as interrupções geradas pelo *timer* do que servindo à outras funções. Assim, cada objeto que instancia (ou usa) um *timer*, como o Alarm, Scheduler e Chronometer, nunca realmente tocam em algum registrador do *timer* (portanto esses componentes são independentes de arquitetura), e, no lugar disso, computam quantos ticks, isto é, quantas interrupções de *timer* aconteceram.

Para exemplificar o funcionamento destes componentes, tomemos o escalonador de processos. No construtor do escalonador, é enviado como parâmetro o periodo de escalonamento, ou seja, quanto tempo (no máximo) uma *thread* pode executar antes de ser escalonada. Para se saber quantos ticks devem ser contados antes de se escalonar um processo, basta dividir a frequência em que os ticks incrementam, pela frequência de escalonamento.

Por exemplo, se o *timer* gera uma interrupção a cada 1 milisegundo (1000 Hz), e o escalonador escalona um processo a cada 10 milisegundos (100 Hz), o número de ticks a se contar é $1000/100 = 10$ ticks. Estes são os valores usados na implementação também.

No caso do Alarm em específico, internamente há uma fila com todas as requisições de alarme, ordenado do menor tick ao maior. Quando ocorre uma interrupção de *timer*, é chamado primeiramente o *handler* que gerencia esta fila, e, caso um alarme desta fila já tenha esperado os ticks que requisitou, então o *handler* desse alarme é chamado (este *handler* é definido pelo usuário que instanciou o alarme).

O construtor do `Zynq_Timer` recebe como parâmetro a frequência que o contador deve contar, assim como o *handler* que deve ser chamado quando esta contagem terminar (isto é, a função chamada quando acontecer uma interrupção devido ao *timer* ter chego a zero), e um número chamado `channel`, que serve para demultiplexar qual *handler* deve ser chamado.

A classe `Zynq_Timer` possui um atributo estático (e portanto único para todas as instâncias) definido como `Zynq_Timer* _channels[CHANNELS]`, onde `CHANNELS` é uma constante com o número de diferentes classes usando o *timer* (Scheduler e Alarm). Este vetor é necessário pois, quando uma interrupção de *timer* acontece e o *handler* do *timer* é chamado, este *handler* pode iterar sobre ele, chamando todos os respectivos *handlers* daquelas classes. Abaixo segue um exemplo de código de como isto pode ser feito:

```
Zynq_Timer * Zynq_Timer::_channels[CHANNELS];

void Zynq_Timer::int_handler(IC::Interrupt_Id id)
{
  if((!Traits<System>::multicore || Machine::cpu_id() == 0)
    && _channels[ALARM])
  {
    _channels[ALARM]->_handler();
  }

  if(_channels[SCHEDULER] &&
    (--_channels[SCHEDULER]->_current[Machine::cpu_id()] <= 0))
  {
    _channels[SCHEDULER]->_current[Machine::cpu_id()] =
      _channels[SCHEDULER]->_initial;
    _channels[SCHEDULER]->_handler();
  }
}
```

No código, `_current` guarda o número de ticks atual daquele *timer* (para cada processador), e `_initial` guarda o número inicial de ticks que foi configurado na inicialização do do escalonador. Note então que a cada interrupção de *timer*, o tratador do alarme será chamado (no caso do primeiro

processador), já o do escalonador, somente a cada certo número de ticks ele será chamado (cerca de 10 na implementação).

Os 4 principais registradores a se trabalhar para configurar o *timer* são o `load`, registrador onde se escreve por quantos ciclos se deve contar; o `counter`, que é o registrador que contém o atual valor contado, sendo decrementado a cada ciclo até chegar em zero, chegando em zero o é gerada uma interrupção número 29; registrador `control`, que permite configurar certos comportamentos do *timer*, como o de ativa-lo, ativar modo cíclico, ativar interrupções e atribuir um valor para o `prescale`; e finalmente o registrador `interrupt status`, que, como o nome indica, permite que se leia o status das interrupções de timer. Todos os timers trabalham à metade do *clock* do sistema, ou seja, usando o *clock* CPU_3x2x.

Durante a inicialização do sistema, o *timer* é configurado para gerar interrupções periodicamente, e esta configuração não é alterada durante a execução da aplicação. Como o construtor do `Zynq_Timer` recebe uma frequência como parâmetro, é necessário converter esta frequência para um número de ciclos a se contar. Para isto, é necessário levar em conta a frequência com que o contador é decrementado, para então se definir um valor a ser decrementado periodicamente de modo a fornecer a frequência desejada.

Como sabemos que o *clock* ao qual o *timer* está submetido é metade do *clock* do sistema, e que antes dele chegar ao contador, este mesmo *clock* é dividido por um divisor chamado `prescaler` (que divide pelo valor configurado nele mais 1), podemos dizer a frequência F_dec de decremento do contador é de:

$$\text{F\_dec} = \frac{\text{CPU\_6x4x}}{2 \times (PRESCALER + 1)} \tag{5.2}$$

Logo, usando a mesma linha de raciocínio exposta no exemplo de cálculo de ticks, temos que o valor a ser carregado no `load_register` (que será chamado de `load_value`), sendo F a frequência que se deseja configurar o *timer*, é:

$$\text{load\_value} = \frac{\text{CPU\_6x4x}}{2 \times (PRESCALER + 1) \times F} \tag{5.3}$$

Ou de maneira mais simples:

$$\text{load\_value} = \frac{\text{F\_dec}}{F} \tag{5.4}$$

Precisamos agora definir o `prescaler`. Definimos ele como a razão entre o *clock* do sistema pela frequência desejada (prescaler $= \frac{\text{clock}}{2 \times F}$). Há a premissa de que a frequência desejada não será maior que a do *clock* do *timer*, pois é impossível contar mais rápido que isto. Normalmente esta razão ($\frac{\text{clock}}{2 \times F}$) será

um número maior que 255, já que o clock costuma ser muito mais rápido, e como o campo onde se registra o valor do prescaler possui apenas 8 bits, frequentemente o prescaler será 255.

5.3 MAPEAMENTO DE MEMÓRIA

Por padrão, as 8 primeiras palavras da memória (ou seja, os primeiros $8 \times 4 = 32$ bytes) devem possuir instruções específicas. A primeira palavra (memória posição 0) contém a primeira instrução a ser executada, e, nas 7 palavras seguintes, fica a tabela de vetores (*vector table*). Como abaixo da instrução inicial há uma tabela que não se deseja executar no momento de inicialização do sistema, esta primeira instrução necessariamente é um *jump* para uma outra região, para aí então se iniciar o processo de *boot*. As demais 7 palavras, pertencentes à tabela de vetores possuem, similarmente, *jumps* para o código onde o tratador da exceção se localiza. A primeira instrução da tabela (posição 0x4) deve conter um *jump* o tratador de uma exceção do tipo undefined instruction, depois, nas próximas palavras, a software interruption, prefectch abort, data abort, reserved, irq e, finalmente, fiq, nesta ordem. Vide seções 3.3.1 e 3.2 para mais detalhes.

Desmontando-se o binário da imagem produzida na compilação do EPOS (*dump*), deve-se obter uma saída semelhante a esta exemplificada abaixo em seus primeiros 32 bytes:

```
00000000 <_vector_table>:
 0:  ldr pc, [pc, #2044] ; 804 <_start_addr>
 4:  ldr pc, [pc, #2044] ; 808 <_undefined_instruction_addr>
 8:  ldr pc, [pc, #2044] ; 80c <_software_interrupt_addr>
 c:  ldr pc, [pc, #2044] ; 810 <_prefetch_abort_addr>
10:  ldr pc, [pc, #2044] ; 814 <_data_abort_addr>
14:  ldr pc, [pc, #2044] ; 818 <_reserved_addr>
18:  ldr pc, [pc, #2044] ; 81c <_irq_handler_addr>
1c:  ldr pc, [pc, #2044] ; 820 <_fiq_handler_addr>
```

Também foi necessário definir as pilhas (*stacks*) do sistema, assim como reservar um espaço para a pilha dos tratadores de interrupção. O *layout* escolhido segue na tabela 3. Este é o *layout* usado durante o desenvolvimento. Somente as pilhas de usuário e de algum modo privilegiado (como o supervisor) são necessárias, portanto a pilha de supervisor acabaria por englobar as demais posições que aparecem na tabela 3. Na próxima seção é explicado como foi feito para não precisar reservar espaço na memória para as demais pilhas.

Lembrando que pilhas, num sistema operacional, tradicionalmente crescem em direção à posições menores da memória, por isto que, por exemplo, a pilha Irq possui 64 bytes, já que $\texttt{0x100040} - \texttt{0x100000} = 40_{16} = 64_{10}$. A pilha do usuário, portanto, localiza-se na última posição da memória (512 MB neste caso) e cresce para "baixo" (posições menores de memória) a partir de lá.

A tabela 4 ilustra o restante do mapeamento de memória feito. APP_DATA é a posição onde será armazenado a seção de dados do EPOS, sendo que 1KB é seguramente mais do que o suficiente para esta seção. É possível descobrir qual é o tamanho da seção de dados em um binário através do comando `size <binary>`. Com este comando, observou-se que, na atual versão do EPOS, a seção de dados possui 436 bytes, e a .bss 188 bytes.

System Info é uma `struct` que guarda informações sobre o SO, que é usado durante a inicialização do sistema (e pode ser descartado mais tarde).

Para a MMU, é necessário 3600 KBs para mapear todo o espaço de endereçamento virtual (vide a seção 5.5.2 para entender porque apenas 3600 KBs); a tabela e páginas usada pela MMU está indicada como "Tabela MMU" na tabela 4. "Livre" indica memória não usada. Esta porção não pode ser alocada pois a tabela da MMU precisa iniciar numa posição de memória alinhada à 16 KB.

SYS_HEAP é a *heap* do sistema operacional. Note que as pilhas das *threads* ficam dentro da *heap* do sistema, por isto ele deve ter um tamanho adequado para poder alocar memória suficiente para o número de *threads* que espera-se executar.

Estes valores são editáveis nos `traits`, entretanto também foi necessário atualizar o arquivo `./include/mach/zynq/memory_map.h`.

Pilha	Endereço base	Tamanho máximo (bytes)
Supervisor (CPU1)	`0x00080000`	500784
Supervisor (CPU0)	`0x00100000`	524288
Irq	`0x00100040`	64
System	`0x00100080`	64
Abort	`0x001000c0`	64
Fiq	`0x00100100`	64
Undefined	`0x00100140`	64

Tabela 3 – Pilhas do sistema com seus tamanhos e posições.

Dado	Endereço base	Tamanho Alocado
APP_DATA	0x00100140	1KB
System Info	0x00100540	260 bytes
Livre	0x00100644	14KB
Tabela MMU	0x00104000	3600KB
SYS_HEAP	0x00488000	128MB

Tabela 4 – Mapeamento de memória.

5.4 CONTROLADOR DE INTERRUPÇÕES

Para se usar o controlador de interrupções (que será referenciado como GIC no restante desta seção, de *Generic Interrupt Controller*), é necessário antes inicializar o distribuidor de interrupções e as interfaces dos processadores.

É no distribuidor que é determinada a prioridade de cada interrupção, e onde é decidido se determinada interrupção deve ou não ser encaminhada para a interface de um determinado processador. Todas as interrupções passam por ele.

A interface dos processadores é por onde os processadores se comunicam com o GIC. Nela o processador pode confirmar que recebeu a interrupção (*acknowledge*), indicar que terminou de tratar a mesma, definir prioridades entre diferentes interrupções, indicar uma política de preempção de interrupções, ou mesmo desligar esta interface. Como o GIC é dividido logicamente é ilustrado na imagem 12.

5.4.1 Inicialização

Na inicialização do distribuidor, através dos registradores mapeados em memória de configuração do mesmo, é definido, para cada uma das possíveis interrupções, se elas são *level-sensitive* ou *edge-triggered*. Em seguida configura-se a prioridade de cada interrupção. A princípio todas as interrupções foram definidas como tendo a mesma prioridade, mas isto é configurável caso necessário. É configurado então o processador-alvo de cada interrupção, isto é, para quais interfaces de processador uma determinada interrupção será encaminhada. Finalmente então são ativadas as interrupções (ARM. . . , 2008).

A inicialização da interface do processador é mais simples. Primeiro se configura a máscara de prioridade da CPU, isto é, qual é o nível de priori-

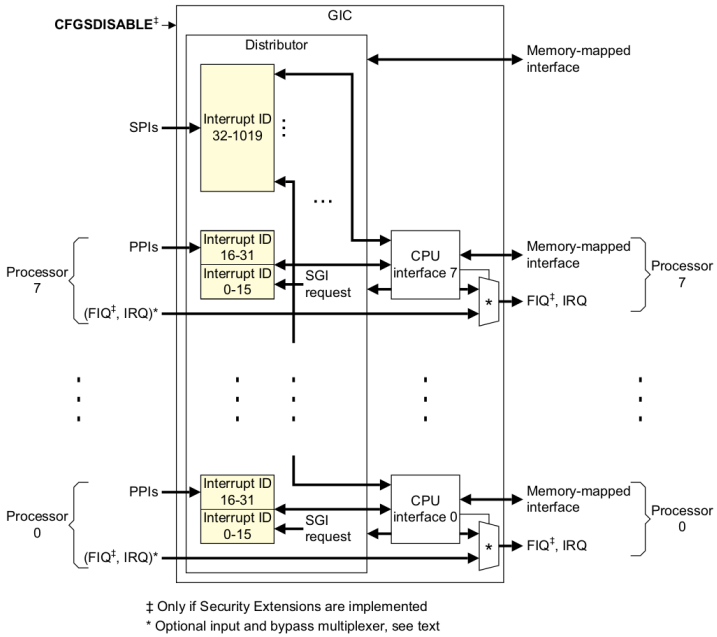

Figura 12 – Divisão lógica do GIC.

dade mínimo que uma interrupção precisa ter para interromper aquele processador. Na implementação, esta máscara está desativada. Depois configura-se a política de grupos de preempção. No GIC é possível separar interrupções em grupos de preempção, onde é definido se determinado grupo pode preemptar determinado outro grupo. Todas as interrupções foram colocadas no mesmo grupo e interrupções podem ser preemptadas.

Adicionalmente a estas configurações, é possível de mascarar as interrupções FIQ e IRQ através do CPSR (*Current Program Status Register*), alterando-se os bits 6 e 7 dele, para mascarar interrupções FIQ e IRQ, respectivamente. Normalmente é o que é feito quando é necessário mascarar as interrupções temporariamente, enquanto se mantém as configurações do GIC.

5.4.2 Fluxo de execução ao se receber uma interrupção

O processador, ao receber uma interrupção, (portanto as interrupções estão ativas e não mascaradas pela interface ou pelo CPSR), para a execução do código que estava executando e então executa a instrução contida na tabela

de vetores (mostrada na página 62) correspondente ao tipo de interrupção recebida. Esta instrução é um *jump* para um tratador (*handler*) daquele tipo de interrupção.

O principal tratador é o tratador de interrupções IRQ (`irc_handler`), sendo que este precisa ser discutido com mais profundidade. Abaixo segue o código deste tratador:

```
43 void _irq_handler() {
44   ASMV(
45   // A few definitions
46   ".equ ARM_MODE_FIQ,       0x11 \n"
47   ".equ ARM_MODE_IRQ,       0x12 \n"
48   ".equ ARM_MODE_SVC,       0x13 \n"
49   ".equ IRQ_BIT,            0x80 \n"
50   ".equ FIQ_BIT,            0x40 \n"
51   // go to SVC
52   "msr cpsr_c, #ARM_MODE_SVC | IRQ_BIT | FIQ_BIT \n"
53   //save current context (lr, sp and spsr are banked
    registers)
54   "stmfd sp!, {r0-r3,r12,lr,pc}\n"
55
56   "msr cpsr_c, #ARM_MODE_IRQ | IRQ_BIT | FIQ_BIT\n"//go to
    IRQ
57
58   "sub r0, lr, #4 \n" // return from irq addr
59   "mrs r1, spsr   \n" // pass irq_spsr to svc r1
60   //go back to SVC
61   "msr cpsr_c, #ARM_MODE_SVC | IRQ_BIT | FIQ_BIT\n"
62   "add r2, sp, #24 \n" //sp+24 is the position of the saved
    pc
63
64   // save return address into the pc position
65   "str r0, [r2] \n"
66   "stmfd sp!, {r1} \n"    // save irq-spsr
67
68   );

71   IC::int_handler();
72
73   ASMV(
74   "ldmfd sp!, {r0}              \n"
75   "msr spsr_cfxs, r0\n"//restore IRQ's spsr value
76                        //to SVC's spsr
77   "ldmfd sp!, {r0-r3,r12,lr,pc}^ \n" // restore context
78   //the ^ in the end of the above instruction makes the
79   //spsr to be restored into svc_cpsr
80   );
81 }
```

Após selecionar qual handler chamar do vetor de exceções, o proces-

sador muda de modo, indo, no caso de uma interrupção IRQ, para o modo de execução IRQ. Neste modo há 3 registradores banqueados: O SPSR, que contém o valor do registrador CPSR imediatamente antes da interrupção, sendo necessário para que seja possível restaurar o valor original do CPSR após tratar a interupção; o LR (*link register*), que contém o endereço da próxima instrução que seria executada imediatamente antes da interrupção mais 4; e, finalmente, o SP (*stack pointer*), que aponta para a pilha própria desde modo (cada modo pode possuir sua própria pilha).

Para evitar desperdício de memória reservando uma pilha própria apenas para este modo, optou-se por não usar uma pilha no modo IRQ, e, no lugar disto, usar sempre a mesma pilha do modo supervisor (que é o modo de execução do processador quando ele inicia), isto também permite um melhor gerenciamento da pilha no caso de preemptação. Para isto, a primeira instrução a executar é uma mudança de modo para voltar ao modo supervisor, enquanto mantendo novas interrupções desligadas; lá é salvo o contexto na pilha daquele modo. Entretanto para que seja possível restaurar o fluxo de execução no mesmo estado em que o processador estava no momento imediatamente antes da interrupção, é necessário voltar ao modo IRQ para obter-se os valores contidos dos registradores banqueados SPSR e LR; após isto, pode-se então voltar ao modo *supervisor*. Na linha 62 do código é somado 24 à pilha pois lá é a posição de memória onde está salvo o PC após ele ter sido empilhado na linha 54, e deseja-se sobreescrever aquele valor do PC pelo valor que estava contido no LR do modo IRQ (menos 4), pois aquela é a próxima instrução que seria executada antes da interrupção. Feito isto, salva-se o valor do SPSR do modo IRQ no topo da pilha (que é o valor do CPSR pré-interrupção), para ser restaurado ao CPSR mais tarde.

Agora que o contexto foi salvo corretamente para ser restaurado, pode-se então chamar um tratador de interrupções genérico (que será discutido mais a frente) e escrito em C++. Após o `int_handler` da linha 71 retornar, é feita a restauração do contexto. Primeiro se salva o CPSR salvo na pilha no SPSR, na última instrução (`ldmfd`), na forma em que ela está (com um ^ no final dela e com o PC na lista), ela automaticamente restaurará o valor que está no SPSR para o CPSR. Como o PC está na lista, o fluxo de execução terá retornado a executar as instruções de antes da interrupção ocorrer.

Agora será discutido como que as interrupções são tratadas individualmente. O corpo do `int_handler` é bastante curto, então vale a pena escreve-lo aqui:

```
void Zynq_IC::int_handler()
{
    unsigned int icciar_value = CPU::in32(IC::GIC_PROC_INTERFACE
        | IC::ICCIAR);
    IC::Interrupt_Id id = icciar_value & IC::INTERRUPT_MASK;
```

```
    if(id == 1023){
        kout << "Spurious interruption received\n";
        return;
    }
    _vector[id](id);
    CPU::out32(IC::GIC_PROC_INTERFACE | IC::ICCEOI, icciar_value
        );
}
```

A primeira coisa que o tratador faz é descobrir qual é o número da interrupção que foi gerada, para assim saber como tratar aquela interrupção. Isto é feito lendo-se o registrador ICCIAR (*Interrupt Acknowledge Register*), que provê o número da interrupção e também o processador endereçado. É possível que uma interrupção já tenha sido tratada por outro processador, quando isto acontece, o GIC emite uma *Spurious Interruption* para indicar isto. Quando se detecta isto, o tratador não precisa tomar nenhuma outra ação, basta retornar a execução normal.

Com o número da interrupção em mãos, pode-se então chamar o tratador daquele tipo de interrupção. O `_vector` é um vetor de *handlers*, onde para cada posição *i*, existe o tratador da interrupção número *i*. Para sinalizar que uma interrupção foi tratada, deve-se escrever no registrador ICCEOI (*End of Interruption*) o número lido no ICCIAR (ou seja, o número da interrupção e processador de destino).

Normalmente, interrupções de *timer* são o tipo mais frequente de interrupção durante a execução do sistema. Dele dependem o escalonador de processos (ou *threads*), Delay, Chronometer e Alarm. Como mencionado na seção do porte do *timer*, uma mesma interrupção pode gerar a chamada de mais de um handler, como a do *timer* que chama a do Alarm e do escalonador. Com isto fica ilustrado que uma única interrupção pode gerar a chamada de vários tratadores para esta mesma interrupção.

5.5 MMU

As principais funções de uma MMU (*Memory Managemend Unit*) são proteção de memória, ou seja, não permitir acesso a certas regiões da memória por processos não autorizados, e de fazer o mapeamento de memória virtual para memória física, mapeamento este que facilita a escrita de aplicações já que o desenvolvedor dela não precisará estar ciente de como a memória é mapeada internamente pelo sistema operacional.

A MMU consegue cumprir estes dois objetivos (e outros mais) através do uso de uma Tabela de Tradução de Páginas (que chamaremos de TTP),

onde, a grosso modo, cada linha desta tabela é indexada pelo valor do endereço virtual, e na entrada correspondente há o endereço físico assim como *flags* que indicam, dentre outras coisas, se aquela região pode ou não ser acessada pela aplicação em execução.

Esta tabela é salva na memória principal (RAM) do sistema, e configurar a MMU significa especialmente criar métodos para gerenciar e popular esta tabela. Para entender como isto é feito, será explicado agora como é estruturada esta tabela, como que ocorre a tradução de memória virtual para física, e o que cada entrada da tabela deve ter.

5.5.1 Estrutura da Tabela de Tradução de Página

A TTP possui dois níveis, portanto, desconsiderando-se a TLB, é necessário dois acessos à TTP para traduzir um endereço, caso configurado para páginas de 4 KB. a MMU suporta páginas de 4 KB e 64 KB, assim como seções de 1 MB e 16 MB. Como cada endereço virtual corresponde à exatamente uma entrada na TTP, usar páginas grandes reduz o tamanho da TTP (consequentemente o uso de memória por ela), entretanto usar páginas menores (4 KB) melhoram muito a eficiência da alocação dinâmica de memória e desfragmentação (ZYNQ-7000..., 2014, p. 77), porém mapear um espaço de 4 GB exigiria milhões de entradas na tabela. Foi justamente para conciliar estes fatores, que decidiu-se por uma tabela de dois níveis, como será melhor ilustrado a seguir.

A tabela de tradução páginas nível um (TTP1), também chamada de "*master table*", divide todo o espaço de endereçamento de 4 GB em 4096 seções de 1 MB, portanto, como cada entrada desta tabela tem o tamanho de uma palavra, seu tamanho total em memória é de $4 \times 4096 = 16$ KB. Os 2 bits menos significativos (lsb) de cada entrada desta tabela, definem que tipo de entrada ela é, que pode ser um dos 5 tipos:

- Bits lsb 00: É uma "fault entry", e gera uma exceção do tipo *prefetch* ou *data abort*, dependendo do tipo de acesso. Este tipo de entrada indica que não há mapeamento virtual para esta região.

- Bits lsb 01: Indica a posição de memória de uma tabela de páginas (nível 2).

- Bits lsb 10 com bit 18 em 0: Aponta para uma seção de 1 MB.

- Bits lsb 10 com bit 18 em 1: Indica uma seção de 16 MB, este tipo de entrada exige 16 posições na TTP1.

- Bits lsb 11: Reservado, gera faltas de tradução e não deve ser usado.

A figura 13 ilustra o formato de uma entrada da TTP1. NS significa *non-secure*, e só tem semântica com extensões de segurança habilitadas (fora do escopo do trabalho). SBZ significa *should be zero*. Bits C, B, TEX e AP serão explicados após a explicação do formato de uma entrada da TTP2, pois esta tem campos análogos.

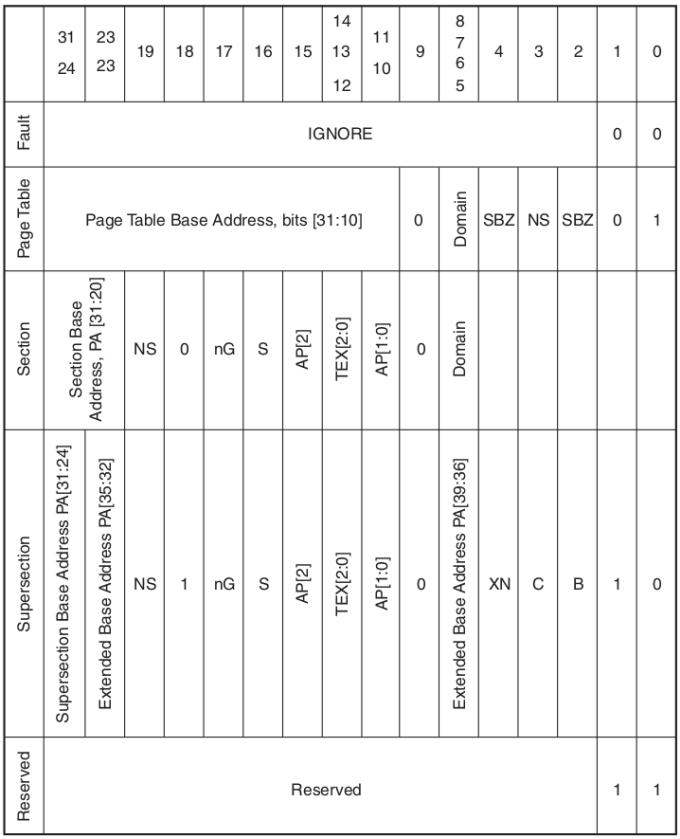

Figura 13 – Formato de uma entrada da tabela de tradução de páginas nível 1.

Vamos agora entender como se traduz de um endereço virtual, o endereço de uma TTP2. No coprocessador CP15, existe um registrador chamado TTBR (dois na realidade, TTBR0 e TTBR1), sigla de *Translate Table Base Register*, onde é guardado o endereço da primeira posição da TTP1. Com a MMU ligada, quando há um acesso à memória, automaticamente a MMU

pulará para a posição indicada no TTBR, e indexará esta tabela usando os 12 bits mais significativos do endereço virtual. A maneira exata de como é feito este cálculo é mostrada na figura 14.

Na posição de memória encontrada no passo anterior, estará uma entrada da TTP1 no formato indicado na figura 13. No caso de uma entrada que indica uma outra TTP, os primeiros 22 bits indicam o endereço onde estará esta tabela, com os demais bits em 0, com exceção do último e os do campo de domínio (que será explicado a seguir). Com estas informações, a MMU já pode localizar a posição da TTP2. Para saber qual das entradas da TTP2 deve ser selecionada, é usado os bits [19:12] do endereço virtual para indexar a TTP2.

Os 4 bits do campo de domínio indicam em qual dos possíveis 16 domínios de memória aquela região se encontra, um domínio é apenas um conjunto de regiões de memória. A utilidade disto é que se pode, para cada domínio, especificar como será o controle de acesso nele. Cada domínio pode estar configurado (pelo registrador DACR) como um dos três tipos de acesso: *Acesso não permitido*, qualquer acesso a esta região gera uma falta de domínio; *Cliente*, permissões de acesso são verificadas, e podem gerar uma falta de permissão; e *Administradores*, onde permissões de acesso não são verificadas (bits AP e APx são ignorados).

A TTP2 possui 256 entradas, cada uma com o tamanho de uma palavra, logo, cada TTP2 precisa de 1 KB de memória. Note que como uma entrada da TTP1 provê 22 bits para endereçar uma TTP2, esta TTP2 pode efetivamente estar localizada em qualquer região da memória, já que 22 bits são suficientes para indexar qualquer região de 1 KB de memória.

A figura 15 ilustra como é o formato de uma entrada na TTP2. Os 2 bits menos significativos indicam se a página possui 4 KB, 16 KB ou se a região não está mapeada. Com o endereço de uma entrada da TTP2 em mãos, basta combinar os 20 bits mais significativos desta entrada com os 12 bits menos significativos do endereço virtual, e teremos traduzido o endereço virtual para um físico.

Agora vamos discutir o que cada campo de uma entrada da TTP2 significa.

AP e APx: São os bits que codificam o tipo de acesso que aquela porção de memória possui. Estes bits só são checados caso do domínio seja de *Cliente*. Um acesso que não possui as permissões necessárias geram uma exceção do tipo *prefetch* ou *data abort*. APx e AP[2] são sinônimos. A tabela 5 ilustra os diferentes tipos de acesso que podem ser codificados.

TEX: Controla políticas de compartilhamento de memória, caso aquela região seja compartilhada.

C e B: Estes campos combinados controlam a política de "cachea-

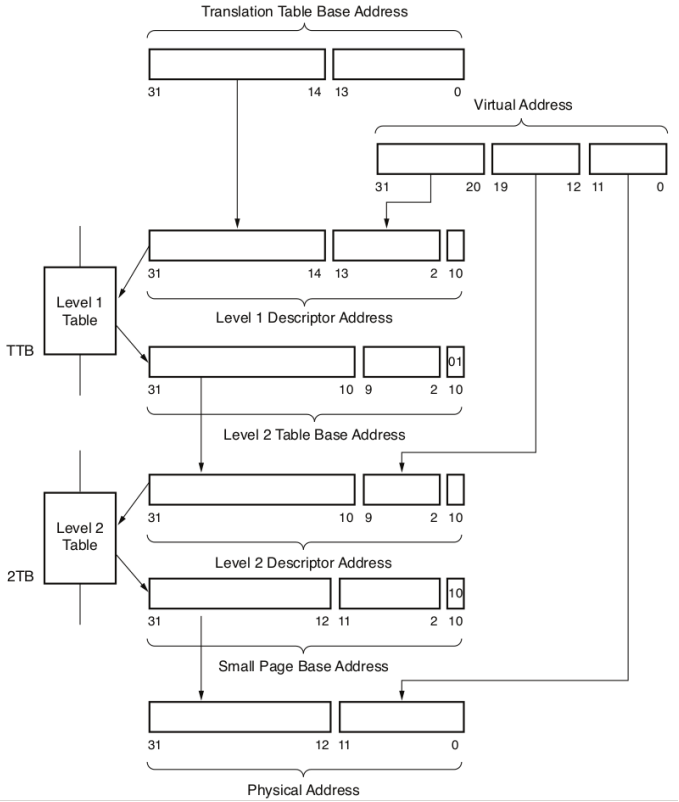

Figura 14 – Processo de tradução de memória virtual para física.

mento" daquela região de memória, ou seja, se ela pode ser salva na cache, e se pode, qual o algoritmo a a ser usado, como *write-back* ou *write-though*, e também políticas de *write-allocate*.

S: Bit que determina que aquela porção de memória é compartilhada entre processadores.

nG: Responsável pela proteção de memória intraprocessos. Uma região marcada como não global (nG = 1) tem associado à ela um número que o SO atribui a cada processo. Isto permite que a TLP possa ter diferentes mapeamentos simultaneamente sem necessidade de substituir uma entrada.

xN: Bit que marca a região como não executável. É importante por questões de segurança, já que o *fetch* especulativo de instruções pode ler uma porção de memória que possui dados sensíveis.

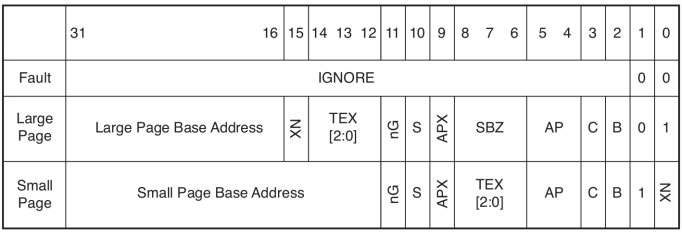

Figura 15 – Processo de tradução de memória virtual para física.

APx	AP[1]	AP[0]	Privilegiado	Não Privilegiado
0	0	0	Sem acesso	Sem acesso
0	0	1	Escrita/Leitura	Sem acesso
0	1	0	Escrita/Leitura	Sem acesso
0	1	1	Escrita/Leitura	Sem acesso
1	0	0	~	~
1	0	1	Leitura	Sem acesso
1	1	0	Leitura	Leitura
1	1	1	~	~

Tabela 5 – Codificações das permissões de acesso.

5.5.2 Inicializando a Tabela de Páginas

Agora que sabemos como as TTPs são estruturadas, bem como o formato de cada entrada dela, podemos finalmente popular esta tabela. Há várias estratégias para fazer isto (EMBEDDED. . . , 2014), dependendo da forma desejada de mapeamento, por exemplo:

- Mapeamento por Demanda: Pode-se fazer com que determinado endereço virtual não possua um mapeamento correspondente a priori, gerando uma exceção que é tratada criando-se uma TTP2 que mapeie aquela área.

- Mapeamento 1 para N: Com o advento do bit nG, é possível também mapear uma mesma área de memória virtual em diferentes áreas da memória física, de modo que quando o escalonador de processos escalona num novo processo (ou *thread*), ele também modifica o mapeamento de memória, deste modo, diferentes processos poderiam usar uma mesma determinada posição de memória virtual.

- Mapeamento 1 para 1: É o mapeamento mais simples possível, de modo que a posição de memória virtual X corresponde a posição de memória física X.

No mapeamento adodotado, a princípio mapeia-se em em uma relação 1:1 os primeiros 512MB de memória virtual para os 512MB de memória física disponíveis, ou seja, o endereço virtual será o mesmo que o endereço físico. Entretanto note que com o uso de `Chunks` e `Address_space` é possível alocar uma porção de memória em um mapeamento próprio, como explicado na seção 4.6.

Suponha que a MMU esteja ativa, e deseja-se salvar determinada informação numa determinada posição de memória física P, cujo endereço virtual correspondente seja V. Se apontarmos um ponteiro para a posição P, e escrever nesta posição, a MMU automaticamente irá interpretar P como sendo memória virtual, e irá traduzir P para uma outra posição de memória física qualquer e imprevisível.

Agora suponha que o mapeamento de memória esteja dividido no meio, de tal modo que a segunda metade faz uma relação 1:1 com a memória física. Deste modo, dado um endereço físico P, é possível calcular qual é seu endereço virtual V.

No EPOS foi feito um mapeamento semelhante, o intervalo de memória virtual [0x20000000 : 0x3FFFFFFF] mapeia para [0x0 : 0x1FFFFFFF]. Portanto `P | 0x20000000 == V`, onde | é o operador *or* de bits. Veja o seguinte exemplo de código:

```
static Log_Addr phy2log(Phy_Addr phy){
    return phy | PHY_MEM;
}
static Phy_Addr calloc(unsigned int frames = 1) {
    Phy_Addr phy = alloc(frames);
    memset(phy2log(phy), 0, sizeof(Frame) * frames);
    return phy;
}
```

`memset` irá tentar escrever no endereço enviado como parâmetro, que será interpretado como um endereço virtual, que então será traduzido para um físico correspondente. Entretanto no lugar de um endereço físico, é enviado o endereço lógico no lugar, para que este seja então traduzido para o endereço físico que desejamos. Esta é uma maneira do SO internamente burlar a tradução da MMU.

O restante da memória virtual (1GB-4GB) é traduzido numa relação 1:1. Esta região de memória corresponde principalmente a registradores mapeados em memória, não sendo, portanto, memória RAM. A figura 16 ilustra o mapeamento adotado.

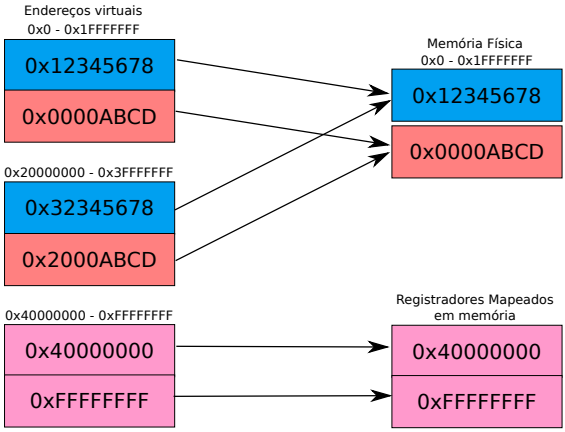

Figura 16 – Mapeamento de memória adotado. Os primeiros 512MB de memória virtual são mapeados 1:1 para a memória física, enquanto os 512MB de memória virtual seguintes mapeiam para os os mesmos 512MB de memória física. De 1GB à 4GB é um mapeamento simples 1:1, referentes aos registradores mapeados em memória.

Note que num mapeamento completo 1:1, a TTP1 tomaria 4×4096 bytes (4KB), e as 4096 TTP2 (uma para cada posição da TTP1) tomariam $4 \times 256 \times 4096$ bytes (4MB). Entretanto como o mapeamento virtual do intervalo 512MB-1GB pode usar as mesmas tabelas TTP2 que foram criadas com o mapeamento do intervalo 0-512MB, pode-se economizar $4 \times 256 \times 512$ bytes. Portanto, no total, as tabelas de tradução de páginas usadas pela MMU usam 3600KB de memória RAM.

Em um mapeamento 1 para 1, para popular as TTPs, primeiramente, em um loop, itera-se 512 vezes (já que há 512 de RAM), salvando em posições sucessivas de uma determinada região da memória os endereços das TTP2s, levando-se em conta as *flags* acima mencionadas, após as 512 posições, itera-se pelas por mais 512 posições, mapeando novamente os primeiros 512 MB de memória física. Após isto, mapeia-se as posições restantes em relação 1:1. Feito isto, é necessário também popular cada TTP2 que foi apontada pela TTP1, em um processo semelhante. Em um mapeamento por demanda, marca-se as regiões da memória virtual como não mapeadas, com a diferença que, ao invés de apenas abortar o acesso, cria-se dinamicamente aquele mapeamento.

Após feita as tabelas, a MMU está pronta para ser ativada. A MMU é controlada pelo coprocessador CP15, é nele que são guardados os regis-

tradores de configuração, endereço base para a TTP, dentre outras funções. Escrevendo-se 1 no bit menos significativo no registrador c1 (SCTLR) deste coprocessador, a MMU é ativada.

5.6 MEDIADOR DA CPU

O mediador da CPU encapsula uma série de funções/rotinas usadas pelo sistema. A maior parte destas funções precisam ser escritas diretamente em assembly, ou usam informações dependente de arquitetura. Este mediador não precisa ser instanciado nem inicializado, ele é apenas um conjunto de funções necessárias por outros componentes.

Entre as funções deste mediador estão:

- Ativar/desativar máscara de interrupções de um dado processador.
- Rotinas para desligar, suspender e reiniciar o sistema.
- Função usada pelo escalonador para fazer a troca de contexto.
- Salvar/alterar registrador que guarda o endereço base da tabela de páginas.
- Funções primitivas usadas em semáforos e mutexes, como tsl (*test and set lock*), finc e fdec.
- Inicializar CPU 1.
- Funções para escrita/leitura de registradores.

Nesta classe também há uma classe interna chamada `Context`. Ela possui funções para salvar e carregar o contexto do processador, e atributos para salvar cada registrador de propósito geral da arquitetura (incluindo sp e pc e cpsr). Esta classe é usada no momento da criação de uma *thread*, e durante a troca de contexto.

5.7 MULTICORE

Ao se ligar a placa, apenas a CPU0 executa código, e é função dela fazer as inicializações necessárias para então ligar o segundo processador (CPU1). Inicialmente, a CPU1 executa uma instrução (`wfe`) que o coloca em modo de espera por evento (*Wait for Event Mode*). Para sair deste modo,

a CPU0 precisa emitir uma instrução do tipo evento de sistema, `sev` (*System Event*). Quando a CPU1 recebe este sinal, ele imediatamente executa a instrução que está no endereço 0xffffffff0, localização onde a CPU0 deve previamente ter escrito uma instrução que faça a CPU1 fazer um `jump` para o ponto de entrada do que se deseja executar com a CPU1. Note que a posição 0xffffffff0 é uma região reservada da memória, e não corresponde à uma posição da RAM (é o mesmo princípio de registradores mapeados em memória).

Isto é suficiente para fazer o segundo processador passar a executar código, entretanto esta é a parte simples, ter dois processadores rodando tem várias implicações sobre como deve ser o *design* de cada componente do sistema, pois agora há a preocupação com coerência e consistência de memória, condições de corrida *racing conditions*) e etc.

Um exemplo de componente que deve que ser adaptado para levar em conta que há mais de um processador, está no código da página 60, pois apenas o CPU0 pode escalonar processos no *design* escolhido. Também é necessário levar em conta o fato de que agora é necessário invalidar a cache privada da CPU1 também, bem como criar uma pilha própria para a CPU1, assim como ativar seu *timer* privado. Através do coprocessador que controla a MMU, é possível de configurar políticas de coerência de memória, como citado na seção 5.5.

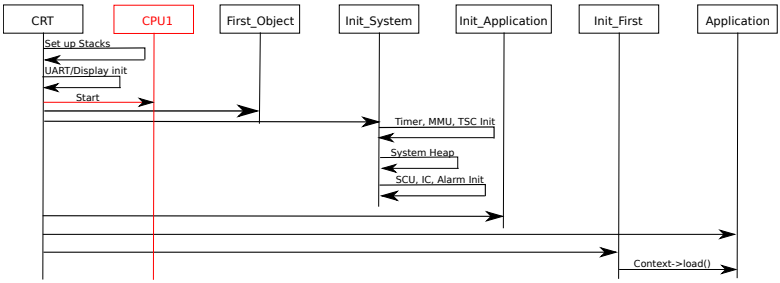

Figura 17 – Ponto de entrada da CPU 1 durante a inicialização.

É necessário também definir um ponto de entrada para o segundo processador. Foi escolhido iniciar o segundo processador já desde os primeiros estágios da inicialização do sistema, como é ilustrado na figura 17. Dentro de classes como `Init_System`, onde são inicializado os mediadores do sistema, há desvios de fluxo (`if's`) para inicializar apenas o que for pertinente para cada processador inicializar. Por exemplo, o *timer* pode ser inicializado para cada *core*, já a MMU não precisa ser inicializada duas vezes.

5.7.1 Simetria

Há duas estratégias comuns de lidar com multicores: Simétrica e assimétrica. Abaixo é definido o significado destes conceitos.

Simétrico:

- Cada processador tem as mesmas funções.
- Único SO para todos os processadores.
- Qualquer processador pode executar tanto a aplicação quanto o SO.
- Processadores precisam ser fisicamente idênticos.

Assimétrico:

- Cada processador possui uma função específica
- Possibilidade de executar mais de um SO no sistema (no máximo 1 por processador).
- SO é executado por um único processador (ou subconjunto de processadores).
- Pode ser implementado em arquiteturas com processadores idênticos ou diferentes.

Note que os itens de cada conceito são flexíveis, podendo haver mistura de alguns itens. Designs assimétricos costumam ser empregados em arquiteturas que possuem processadores difererentes em sua arquitetura, geralmente um deles sendo de menor poder de processamento e consumo de energia, enquanto o outro processador, de melhor desempenho, executa tarefas que exigem maior poder de processamento. Entretanto mesmo com processadores idênticos (no nosso caso, ambos *cores* da Zedboard são idênticos), é possível de designar tarefas diferentes para cada processador, algo bastante usado em computação distribuída.

Na implementação feita, foi escolhido do design simétrico, significando que o EPOS é executado por todos os processadores, assim como a aplicação. O EPOS já possuia certo suporte para isto, pois ele ha havia sido portado para um IA32 *multicore*, implementado de forma simétrica. A decisão sobre qual é o melhor design depende muito do tipo da aplicação, e para as aplicações atuais do EPOS, o modelo simétrico é adequado.

5.8 TESTES

Os testes foram feitos em dois ambientes: Simulação com o virtualizador de sistemas operacionais `qemu`, e com uma Zedboard física.

MMU: Como explicado na seção 5.3, e ilustrado na figura 16, os primeiros 512 MB do espaço de endereçamento virtual são mapeados numa relação 1:1 com a memória física, e então os 512 MB de memória seguintes são mapeados também para os primeiros 512 MB de memória física.

Logo, se a MMU estiver ativa, o endereço virtual X (para $X \in [0..512MB]$) e X+512 MB apontam para a mesma posição de memória física. Portanto, o teste consistiu em, antes de ligar a MMU, escrever um número numa posição de memória X, e então ler as posições X e X+512MB, e notar que evidentemente estas posições possuem valores diferentes. Então a MMU é ligada, e é lido novamente as posições X e X+512 MB, resultando agora no mesmo número, mostrando que a MMU esta ativa e traduzindo corretamente os endereços de memória virtuais.

UART: A UART funcionou sem problemas no `qemu`. Na placa, por via de um cabo usb, leu-se os dados pelo `cutecom`. Os caracteres são enviados com sucesso pelo cabo serial. Pode ocorrer a perda de alguns caracteres dependendo da taxa de transmissão escolhida.

GIC: No porte do GIC, o `qemu` mostrou diferentes comportamentos quando usado diferentes versões, fazendo com que certas configurações do GIC funcionassem apenas em algumas versões. Foi possível analisar o comportamento das exceções e dos seus tratadores, entretanto as interrupções não pareciam estar sendo geradas nos intervalos corretos configurados no `timer`, o que levou à conclusão que este componente só pode ser validado com precisão na placa física. Entretanto o GIC se mostrou não responsível na Zedboard, independente da configuração tentada. Foi feito um relatório a respeito, onde é exposto como, dentro das instruções do manual, este comportamento não pode corresponder ao esperado de acordo com o manual, indicando a possibilidade dele estar incompleto. Será necessário analisar outros códigos que usam a Zedboard.

Timer: O `timer` foi validado usando um `Chronometer`, para contar por uma certa quantidade de tempo, onde após isto foi lido quanto tempo passou. Devido aos problemas descritos acima, na placa física o `timer` foi testado lendo-se o registrador mapeado em memória referente ao atual número que está sendo contado no `timer`, onde constatou-se que ele de fato está operante e fazendo suas contagens de forma periódica.

Multicore: No `qemu` o segundo core foi ligado de forma bem sucedida, sincronizando sua inicialização nas barreiras que já existiam no EPOS com o primeiro *core*, inicializando apenas os mediadores que eram neces-

sários, e finalmente criando e executando suas *threads* (thread `main` para o processador 0, e uma `idle` no caso do processador 1). Como o segundo *core* é ativado através de uma interrupção vinda da instrução `sev`, as dificuldades com o GIC impediram a análise do comportamento na placa.

6 CONCLUSÃO

A necessidade de um sistema operacional de tempo real de possuir suporte para uma plataforma embarcada *multicore* motivou este trabalho, que objetivou implementar o suporte do EPOS para uma plataforma embarcada *multicore*, a Zedboard. Neste trabalho foi discutido como se dá a interação software-hardware do sistema operacional com a placa, através do uso de mediadores de hardware, tanto no projeto do EPOS, quanto do interfaceamento fornecido pela placa. Em particular, a implementação dos mediadores de hardware do EPOS foi descrita.

O EPOS é o o primeiro RTOS de código aberto a suportar os escalonadores de tempo real global, particionado e agrupado (Gracioli, 2014). Assim sendo, este porte abre várias novas linhas de pesquisa de aplicação para este sistema operacional, em particular na área de escalonadores de tempo real. Uma possível nova aplicação do EPOS é o controle de um quadcóptero. Com os algoritmos de tempo real que estão implementados no EPOS, e com os recursos da Zedboard, ampla pesquisa pode ser feita, e novos horizontes de aplicação se abrem.

Como trabalho futuro, este porte pode servir para uma maior validação do escalonador feito em (Gracioli, 2014), pois a Zedboard possui os recursos de hardware necessários para sua implementação (*lock/unlock* de linhas de cache, contadores de desempenho e capacidade de configurar a CPU-alvo de uma interrupção), e naquele trabalho foi usado um IA32 como plataforma de validação, que é uma arquitetura de menor previsibilidade que a Zedboard, e portanto essa pesquisa teria uma sustentação melhor sendo aplicada nesta placa.

Este trabalho descreveu como foi feito o porte do EPOS para a Zedboard. Este porte possibilita novos cenários de aplicação do EPOS e novas linhas de pesquisas, principalmente na área de escalonadores.

REFERÊNCIAS

ARM Architecture Reference Manual, ARM v7-A and ARM v7-R edition, Errata markup. Arm ddi 0406b_errata_2011_q3 (id120611). [S.l.], December 2011.

ARM Generic Interrupt Controller, Architecture Specification. Version 1.0. [S.l.], 2008.

Booch, G. **Generative Programming**. [S.l.]: Addison-Wesley, 1998. ISBN 085353402.

EMBEDDED Bits. 2014. Disponível em: <http://www.embedded-bits.co.uk/2011/mmutheory/>.

EPOS Builtin Mode. 2013. Disponível em: <http://www.lisha.ufsc.br/teaching/dos/exercises/builtin.html>.

EPOS User Guide. 2014. Disponível em: <http://epos.lisha.ufsc.br/EPOS+User+Guide>.

FISCHER, T. **Transparência Arquitetural de Sistemas Embarcados: 8 a 64 Bits**. [S.l.], December 2013.

Fröhlich, A. A. M. **Application-Oriented Operating Systems**. Tese (Doutorado) — Technische Universität Berlin, 2001.

Gracioli, G. **Real-Time Operating System Support for Multicore Applications**. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2014.

Lee, E. A.; Seshia, S. A. **Introduction to Embedded Systems**. LuLu, 2011. ISBN 9780557708574. Disponível em: <http://leeseshia.org>.

Li, Q. **Real-Time Concepts for Embedded Systems**. [S.l.]: CMP Books, 2003. ISBN 1578201241.

LINUX Magazine. 2010. Disponível em: <http://www.linux-magazine.com/Online/News/Ubuntu-10.04-Alpha-2-Removes-HAL>.

NATIONAL Instruments. 2014. Disponível em: <http://www.ni.com/white-paper/3938/pt/>.

Polpeta, F. V.; Fröhlich, A. A. Hardware mediators: A portability artifact for component-based systems. In: **Embedded and Ubiquitous Computing**. [S.l.: s.n.], 2004.

Tanenbaum, A. S. **Operating Systems: Design and Implementation**. [S.l.]: Prentice Hall, 1997. ISBN 0136386776.

UBM. 2013. Disponível em: <http://e.ubmelectronics.com/2013EmbeddedStudy/index.html>.

UBUNTU Wiki. 2014. Disponível em: <https://wiki.ubuntu.com/Halsectomy>.

XILINX.COM. 2014. Disponível em: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm>.

ZEDBOARD. 2014. Disponível em: <http://www.zedboard.org>.

ZEDBOARD, Hardware's User Guide. Version 2.2. [S.l.], 2014.

ZYNQ-7000 Combined Product Table. [S.l.], 2013.

ZYNQ-7000 Data Sheet. Ds187 (v1.10). [S.l.], January 2014.

ZYNQ-7000 Technical Reference Manual. Ug585 (v1.6.1). [S.l.], September 2013.

ZYNQ-7000 Technical Reference Manual. Ug585 (v1.7). [S.l.], February 2014.