

NCESPARC+ **DEFINIÇÕES DE PROJETO**

- Implementa a unidade inteira da arquitetura SPARC V8
- Compatível com o NP do Multiplus baseado no chipset da Cypress
- Objetiva esconder a latência de operações de acesso à memória remota, manutenção da coerência dos caches e sincronização
- Implementa os modelos de consistência de memória Sequencial e Processor Consistency (ordem de leituras e escritas na memória não precisa ser a ordem do programa)
- Baseado no modelo Coarse-Grain Multithreading
- Utiliza ao máximo as definições já feitas no projeto NCESPARC



NCESPARC+

CARACTERÍSTICAS DA ARQUITETURA

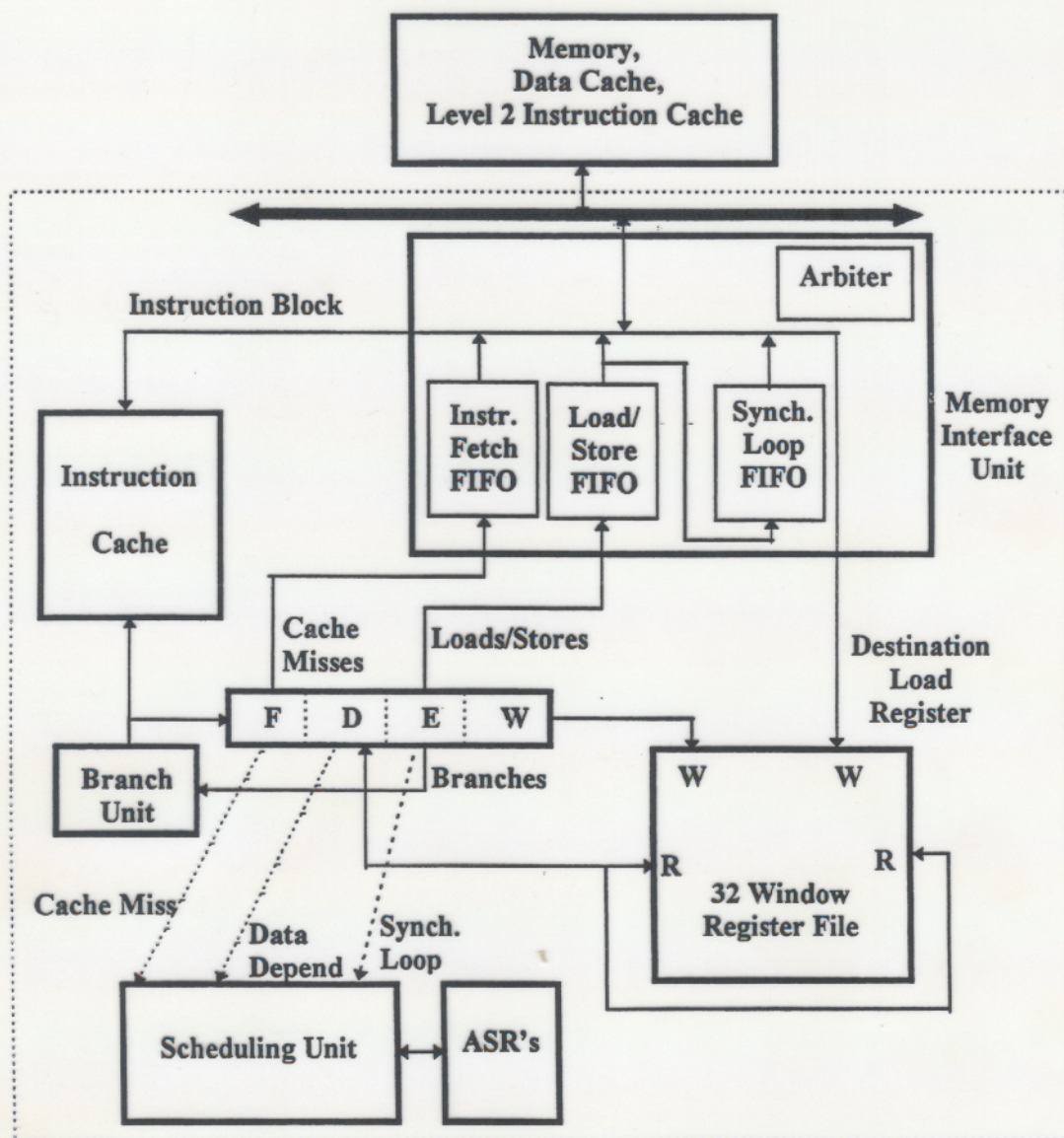
- Suporta um número variável de contextos de hardware (1 a 16)
- Troca de contexto quando:
 - ocorre um miss no cache de instruções
 - há dependência de dados em relação a uma instrução de load pendente
 - ocorre um loop de espera ocupada para sincronização
 - contexto é colocado para dormir por software
- Overhead de troca de contexto ≤ 4 ciclos
- Arquitetura Desacoplada
Pipeline Principal processa instruções de um certo contexto enquanto instruções load/store de outro contexto podem estar ainda pendentes
- Arquitetura dispara instruções em ordem, mas as instruções podem completar fora de ordem (in-order issue, out-of-order completion)

NCESPARC+

PRINCIPAIS COMPONENTES DA ARQUITETURA

- **Cache de Instruções**
 - Mapeamento direto
 - Endereçamento Virtual
 - 16 Kbytes ou 512 blocos de 32 bytes
 - Tempo de acesso inferior a um ciclo de clock
- **Pipeline principal com 4 estágios**
 - Fetch (F), Decode (D), Execute (E) e Write (W)
- **Banco de Registradores**
 - 32 janelas
 - 2 portas de leitura e 2 portas de escrita
 - 2 bits de flag (00: dado ok; 01: dado desatualizado; 10: dado desatualizado -> troca de contexto)
- **Unidade Lógica e Aritmética de 32 bits**
 - Executa 10 operações distintas
- **Barrel Shifter de 32 bits**
 - Shift lógico e aritmético
- **Unidade de Multiplicação/Divisão**
 - Unidade pipelined de execução com 4 ou 5 estágios => instruções completam fora de ordem
 - Usa 2a. porta de escrita no banco de registradores

ARQUITETURA DO NCESPARC+



NCESPARC+

PRINCIPAIS COMPONENTES DA ARQUITETURA

- **Unidade de Escalonamento**
 - Seleciona o próximo contexto a ser associado com o estágio F do pipeline em um ciclo de clock
 - Novo contexto é selecionado em round-robin
 - Um contexto só pode ser selecionado se $ASR[30] = 1$ (há thread mapeada no contexto) e $ASR[31] = 0$ (contexto ready)
 - Operação da Unidade de Escalonamento é inibida se $ASR[29] = 1$ (troca de contexto está desabilitada)
- **Unidade de Branches**
 - Calcula o endereço destino de uma instrução de branch
 - Possui somador próprio para cálculo do endereço
 - Endereço de desvio é passado para o Cache de Instruções no estágio F da instrução seguinte àquela que ocupa o *delay slot*
- **Unidade de Interface com a Memória (MIU)**
 - Realiza os acessos à memória referentes a:
 - busca de instruções com miss no cache
 - operações de load/store
 - loops de sincronização em espera ocupada
 - Implementa protocolo de acesso ao barramento ext.
 - Possui 3 fifo's, uma para cada tipo de acesso
 - Possui um árbitro de acesso ao barramento externo
 - Usa 2a. porta de escrita do banco de registradores

NCESPARC+

CARACTERIZAÇÃO DOS CONTEXTOS

- **Conjunto de Janelas de Registradores**
 - 02 janelas => 16 contextos
 - 04 janelas => 08 contextos
 - 08 janelas => 04 contextos
 - 16 janelas => 02 contextos
 - 32 janelas => 01 contexto

- **Ancillary State Register - ASR [16..31]**
 - bit 31 => indica se o contexto está ready
 - bit 30 => indica se o contexto tem thread mapeada nele
 - bit 29 => desabilita troca de contexto
 - bit 28 => desabilita troca de contexto por miss no cache de instruções
 - bit 27 => desabilita troca de contexto por dependência de dados
 - bit 26 => desabilita troca de contexto em loops de sincronização por espera ocupada

- **Conjunto de Registradores**
 - PC (Program Counter)
 - nPC (next Program Counter)
 - PSR (Processor State Register)
 - TBR (Trap Base Register)
 - WIM (Window Invalid Mask)
 - Y (Multiply/Divide Register)

20/



TROCA DE CONTEXTO POR DEPENDÊNCIA DE DADOS

- Toda instrução de load escreve nos flags do seu registrador destino o código 10 (dado desatualizado) quando se encontra no estágio E do pipeline
- Após o estágio E, a instrução de load é passada pelo pipeline principal para a MIU
- A MIU ao completar a operação de load, atualiza o registrador destino e modifica os seus flags para 00 (dado atualizado)
- Qualquer instrução consulta no estágio D do pipeline os flags dos registradores usados como operandos fonte:
 - flags = 00 => instrução prossegue para est. E
 - flags = 01 => instrução espera no est. D até a conclusão da operação de multiplicação ou divisão
 - flags = 10 => instrução dispara troca de contexto por dependência de dados na Unidade de Escalonamento
- Três ciclos do pipeline são perdidos e duas instruções são anuladas

TROCA DE CONTEXTO POR DEPENDÊNCIA DE DADOS

Ciclo n	Ciclo n+1	Ciclo n+2	Ciclo n+3
Fetch Instr Aritmet.	Decod Instr. Aritmet.	Unid. Escalon. (novo contexto)	Fetch Instr. (novo contexto)
	Fetch Instr. (contexto antigo)	Fetch Inibido Anula 2 instr. (contexto antigo)	

- Unidade de Escalonamento mantém informação sobre o registrador que causou a suspensão de um dado contexto por dependência de dados
- Quando a MIU atualiza o valor de um registrador, essa informação é passada para a Unidade de Escalonamento que acorda, se for o caso, o contexto correto
- Trocas de contexto por dependência de dados são inibidas quando ASR[27] do contexto corrente está ativado

PROCESSAMENTO DA INSTRUÇÃO *LOAD*

- Instruções de *load* são processadas em apenas um ciclo no pipeline principal. Dois ciclos são necessários no processamento de instruções de *load* duplo
- Se o processador estiver operando no modo *Sequential Consistency*, a instrução de *load* é simplesmente enfileirada na fila de *load/store* da MIU ao final do estágio E do pipeline
- Quando o processador está operando no modo *Processor Consistency* (ordem de leitura e escrita na memória pode não ser a ordem do programa), antes de colocar uma instrução de *load* na fila de *load/store*, verifica-se se há uma instrução de *store* ainda pendente na fifo de *load/store* que escreve no mesmo endereço referenciado pela instrução de *load*
- Nesse caso o dado ^{ser} ~~é~~ escrito pelo *store* é retornado como resultado da operação de *load*
- Para implementar essa operação de *by-pass*, a fifo de *load-store* deve se comportar como uma memória associativa => tamanho da fifo de *load/store* não pode ser muito grande

DEPENDÊNCIAS DE SAÍDA

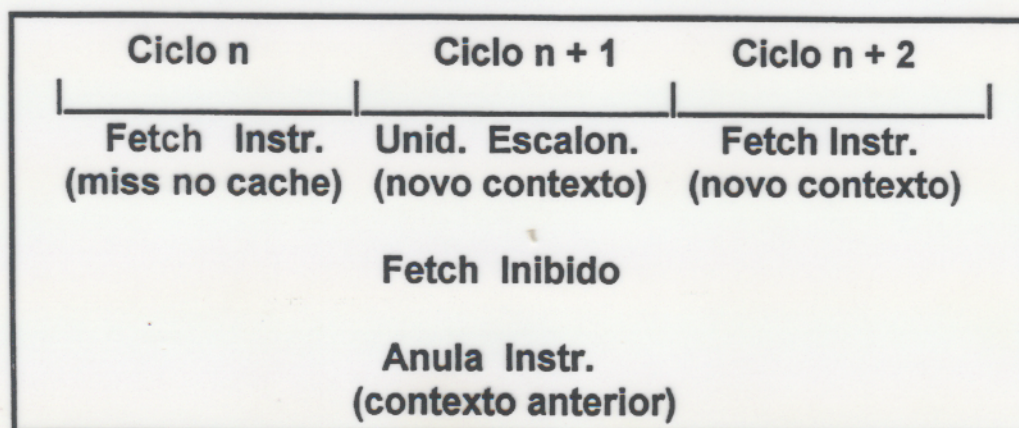
- Como as instruções podem completar fora de ordem é possível ocorrer problemas com dependências de saída
- No exemplo abaixo podemos ter problemas se a instrução de load completa após a instrução *add* e antes da instrução *sub* quando o desvio (*bne*) é tomado:

```
        load R1  
        bne R2, L  
        inc R1;  
        ba L1:  
L:      add R1, R2, R5;  
        inc R2;  
        sub R6, R1, R7;  
L1:     :  
        :
```

- Para evitar esse problema:
 - toda instrução que escreve em registrador com flag 01 ou 10, coloca esse flag em 00.
 - instruções de load só escrevem no registrador se o flag for 10 (configuração definida por ela no estágio E).
 - instruções de multiplicação e divisão só escrevem no registrador se o flag for 01 (configuração definida por elas no estágio E)

TROCA DE CONTEXTO POR MISS NO CACHE DE INSTRUÇÕES

- Miss no cache de instruções é detectado ao final do estágio F do pipeline
- Busca da instrução é passada para a MIU que atualiza um bloco do cache ao final da busca
- A Unidade de Escalonamento gasta mais um ciclo para definir a instrução a ser buscada em relação ao próximo contexto
- Dois ciclos são perdidos e uma instrução é anulada
- Trocas de contexto por miss no cache de instrução são inibidas se $ASR[28] = 1$ no contexto atual



LOOP DE SINCRONIZAÇÃO POR ESPERA OCUPADA

- Possui tipicamente o seguinte formato:

```

loop:    ldstub    [lock], rx
         orcc      r0, rx, r0
         bne       loop
         nop
    
```

- Muitas vezes para reduzir a interferência do loop nos caches de dados faz-se o seguinte procedimento:

```

retry:   ldstub    [lock], rx
         orcc      r0, rx, r0
         be        cont
         nop
loop:    load       [lock], rx
         orcc      r0, rx, r0
         bne       loop
         nop
         ba        retry
cont:    nop
         :
         :
    
```

- Em qualquer dos casos o loop de espera ocupada é formado tipicamente por 4 instruções, sendo que a primeira pode ser *ldstub*, *swap* ou *load* simples

PROCESSAMENTO DE LOOPS DE ESPERA OCUPADA

- O loop é executado pela primeira vez no pipeline principal com o envio da primeira instrução (ldstub, swap ou load) para a MIU
- Instruções de load, ldstub e swap colocam em 1, no estágio E, o valor de um contador associado ao contexto sempre que esse contador tem valor diferente de 4
- As demais instruções do loop (or, bne e nop) incrementam esse contador no estágio E do pipe se o seu valor for, respectivamente, 1, 2 e 3
- Assim, na próxima vez que o loop é executado, as instruções de load, ldstub ou swap encontram o contador em 4 *
- Nesse ponto, é gerada uma instrução interna para a MIU, contendo toda a semântica do loop, e é feita uma troca de contexto (3 instruções são anuladas)
- A instrução interna (*instrução sinc*) é inicialmente armazenada na fifo de load/store da MIU



PROCESSAMENTO DA INSTRUÇÃO INTERNA *SINC* NA MIU

- Quando uma instrução *sinc* atinge a primeira posição da *fifo* de *load/store* ela é processada pela MIU
- Se sua condição de teste for satisfeita, a MIU a retira da fila, atualiza o registrador destino no banco de registradores, atualiza os respectivos flags e informa a Unidade de Escalonamento essa operação
- Se a condição de teste da instrução *sinc* não for satisfeita, a instrução é transferida para o final da fila de sincronização
- Instruções na primeira posição da fila de sincronização nunca têm prioridade de acesso à memória em relação às instruções de *store* na primeira posição da fila de *load/store* para permitir que as instruções de *store* façam eventualmente as liberações dos *locks* esperados
- O *Árbitro da MIU* pode ser configurado para dar ou não prioridade ao tratamento das instruções de *sinc* em relação às instruções de *load* na fila de *load/store*

PROBLEMAS NO PROCESSAMENTO DA FILA DE SINCRONIZAÇÃO

- A fila de sincronização deve ter capacidade para armazenar pelo menos uma instrução de *sinc* por contexto pra evitar ocorrências de *deadlock*
- Apesar disso, o processamento da fila de sincronização, sem maiores cuidados, pode gerar situações de *deadlock*
- Essas situações podem ocorrer, por exemplo, em sincronizações com barreira em que o contexto A realiza um loop de espera ocupada para ver se o evento de chegada de todas as threads na barreira já ocorreu e o contexto B está tentando obter o *lock* para incrementar o contador de chegadas na barreira
- Se a instrução *sinc* do contexto A estiver na frente da instrução *sinc* do contexto B na fila de sincronização e for adotada a política de não remover o elemento da frente da fila até sua condição de término ser satisfeita, ocorrerá *deadlock*
- Implementação da fila de sincronização deve permitir que haja inversão de ordem entre instruções de *sinc* referentes a endereços diferentes na memória

PROBLEMAS PENDENTES

- **Manutenção do Modelo de Exceções Precisas para operações que completam fora de ordem**
- **Concepção da MIU**
- **Concepção da Unidade de Escalonamento**
- **Limitação da Banda Passante de Acessos à Memória**
- **Projeto Lógico Detalhado**
- **Aprendizado dos Sistemas de CAD da Synopsys e Mentor**