

2º Trabalho Prático de Programação Concorrente - 2010

Desenvolva uma aplicação concorrente em C com OpenMPI e no Linux, a qual simule o comportamento de uma CPU MIPS multiciclo, descrita em [1]. Não há necessidade de implementar as instruções *beq* (*branch-on-equal* - desvio condicional) e *j* (*jump* - desvio incondicional). Implemente apenas as instruções: *lw* (*load word*), *sw* (*store word*), *add*, *sub*, *and*, *or* e *slt* (*set-on-less-than*).

Para sua implementação tenha como base o caminho de dados completo e os sinais de controle da CPU MIPS multiciclo, ambos descritos em [1]. O caminho de dados apresentado em [1] deve ser simplificado para implementação deste trabalho, visto que o livro também considera a implementação das instruções *beq* e *j*. Nesta simplificação devem ser removidas, por exemplo, as unidades funcionais responsáveis pelo deslocamento de 2bits à esquerda e o multiplexador que determina a escrita em PC. Há outras simplificações que necessitam ser feitas também e você deverá analisar cada caso e realizá-las quando adequado.

Cada unidade funcional deve ser um processo que executará concorrentemente. Os registradores *PC*, *RI* (Registrador de Instrução), *MDR* (Registrador de Dados da Memória), *A*, *B* e *SaídaALU* também são representados como processos (cada um sendo um processo).

A aplicação deverá ter também processos para: *clock*, *unidade de controle*, *multiplexador* (lembre-se que há 5 multiplexadores no caminho de dados, portanto, devem ter 5 processos apenas para esses multiplexadores), *memória RAM*, *banco de registradores*, *extensão de sinal*, *ALU* e *UC da ALU*.

Estes processos executam independentemente e devem comunicar/sincronizar adequadamente (seguindo primitivas de passagem de mensagens MPI ensinadas em sala de aula) para garantir a semântica da execução da CPU MIPS multiciclo. A geração dos sinais pela UC é o primeiro ponto que requer sincronização após o ciclo ter começado. Outro ponto que requer sincronização é a saída dos multiplexadores, visto que tais saídas devem ser enviadas corretamente para então serem usadas pelas respectivas unidades funcionais.

O processo *clock* é responsável por iniciar um novo *clock*, este simulado através do uso de passagem de mensagens. O processo *clock* também é responsável por atualizar as informações gravadas no ciclo *i-1* para que estas estejam disponíveis no ciclo *i*. Considere sempre que as informações lidas dos elementos de lógica sequencial foram atualizadas no ciclo *i-1*. Considere que não é possível ler uma informação gravada em um elemento de lógica sequencial no mesmo ciclo.

Inicie todos os processos de uma só vez através do *miprun*. Não use a *MPI Comm spawn()* para a geração dos processos. Seu código deve seguir o paradigma SPMD.

Desenvolva a aplicação concorrente segundo as diretrizes passadas aqui e em aula. As questões omissas e/ou ambíguas serão fixadas pelo professor. Qualquer dúvida entre em contato com o professor para a orientação adequada. Tenha uma postura pró-ativa e procure orientação.

O trabalho deverá ser entregue no *moodle*. IMPORTANTE: destaque como comentário no início do código os nomes dos integrantes do grupo *que de fato* participaram do desenvolvimento do trabalho. Os nomes que não aparecerem nessa relação ficarão com nota "zero" no trabalho. A data máxima de entrega deste trabalho é 18/06, às 23:55h. Os detalhes do sistema de avaliação deste trabalho serão disponibilizados aos alunos antes da entrega do mesmo.

Referências

[1] Patterson, D.A., Hennessy, J.L. Computer organization and design : the hardware/software interface, 3ª ed., Elsevier, Amsterdam, 2005.