Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

5º Ano da Licenciatura em Engenharia Informática
Ramo de Computadores e Sistemas
Disciplina de Projecto 2003/2004

IJVM em FPGA

Setembro de 2004

Autor
Rui Filipe Ribeiro Picas de Carvalho

Orientador
Prof. José Carlos Alves

Índice

1 Introdução

1.1 Objectivos

O objectivo deste projecto é implementar um microprocessador baseado na proposta **IJVM** (*Integer Java Virtual Machine*) de Andrew S. Tanenbaum na 4ªa edição do livro "Structured Computer Organization", capaz de executar um subconjunto das instruções da *Java Virtual Machine*.

Pretende-se obter uma implementação do mesmo microprocessador em hardware reconfigurável utilizando para o efeito uma plataforma de desenvolvimento baseada em **FPGA** disponível no mercado, e modelando os seus componentes recorrendo a uma linguagem de descrição de hardware (**Verilog HDL**).

1.2 Motivação

A motivação primordial do projecto foi percorrer os diversos passos envolvidos no processo de concepção, implementação e funcionamento de um microprocessador, seguindo o percurso desde o software até à realização prática e experimentação do hardware.

Além disso, foi também importante perceber a tecnologia de hardware reconfigurável, em especial os dispositivos **FPGA**, e o papel que actualmente desempenham na resolução de problemas que tradicionalmente são implementados em sistemas computacionais "convencionais".

1.3 Organização do documento

O documento encontra-se organizado da seguinte forma:

No Capítulo 2 são apresentados os aspectos mais importantes que caracterizam o microprocessador objecto de implementação.

No Capítulo 3 encontram-se descritos os aspectos relacionados com a implementação propriamente dita. Neste capítulo são enumeradas um conjunto de alterações e/ou adaptações. É igualmente apresentada a plataforma de desenvolvimento e estrutura da implementação.

Finalmente no Capítulo 4 são apresentadas as conclusões do projecto.

2 Mic1 e IJVM

Ao longo deste capítulo é apresentada a organização de um microprocessador capaz de executar um subconjunto do *bytecodes* **Java** segundo o modelo proposto por Andrew S. Tanenbaum em [1]. Este processador é organizado em duas entidades que permitem dissociar a implementação física (microarquitectura) do modelo de programação oferecido ao programador (macroarquitectura).

A macroarquitectura, ou também designada **ISA** (*Instruction Set Architecture*) é a especificação detalhada do conjunto de instruções que um microprocessador é capaz de "entender" e executar. Esta especificação descreve os aspectos do microprocessador que são visíveis do ponto de vista do programador, incluindo tipos de dados, instruções, registos, arquitectura de memória, I/O, ou outras abstracções particulares.

A microarquitectura é a estrutura/organização constituída por um conjunto de elementos e técnicas de suporte à implementação física de uma macroarquitectura.

2.1 Microarquitectura: Mic1

2.1.1 Introdução

A microarquitectura está organizada em dois grandes componentes: o **Data Path** e o **Control Path**.

O **Data Path** é constituído pelo conjunto de componentes que processam a informação durante um ciclo de operação da microarquitectura. É usualmente composto por registos, unidades funcionais (**ALU, Shifter**, etc) e barramentos de interligação.

O **Control Path** consiste numa máquina de estados finitos que permite implementar o controlo do **Data Path**.

2.1.2 Data Path

O diagrama apresentado na Figura 1 representa o *Data Path* do **Mic1** no qual podem ser identificados os seguintes componentes:

- registos: **MAR**, **MDR**, **PC**, **MBR**, **SP**, **LV**, **CPP, TOS**, **OPC** e **H**
- barramentos: **A**, **B** e **C**
- unidades funcionais: **ALU** e **Shifter**

Interessa inicialmente analisar cada um dos componentes do **Data Path** do ponto de vista da topologia. Como será constatado mais à frente o conjunto de ligações de um componente condiciona fortemente o papel que este desempenha.

Relativamente ao conjunto de registos, podem constituídos três grupos:

- Registos de controlo da memória: **MAR**, **MDR**, **PC**, **MBR**
- Registos de uso geral: **SP**, **LV**, **CPP**, **TOS**, **OPC**
- Registo acumulador: **H**

Cada um dos elementos do primeiro grupo tem a particularidade de possuir ligação com o sistema de memória. Este grupo não é uniforme pois a maioria dos seus

elementos não possui o mesmo conjunto de ligações. Em consequência, será interessante descrever o conjunto de ligações de cada elemento.

- **MAR**: memória (*output*); barramento **C** (*input*).
- **MDR**: memória (*input/output*); barramento **C** (*input*); barramento **B** (*output*)
- **PC**: memória (*output*); barramento **C** (*input*); barramento B (*output*).
- **MBR**: memória (*input*); barramento **B** (2 x *output*).

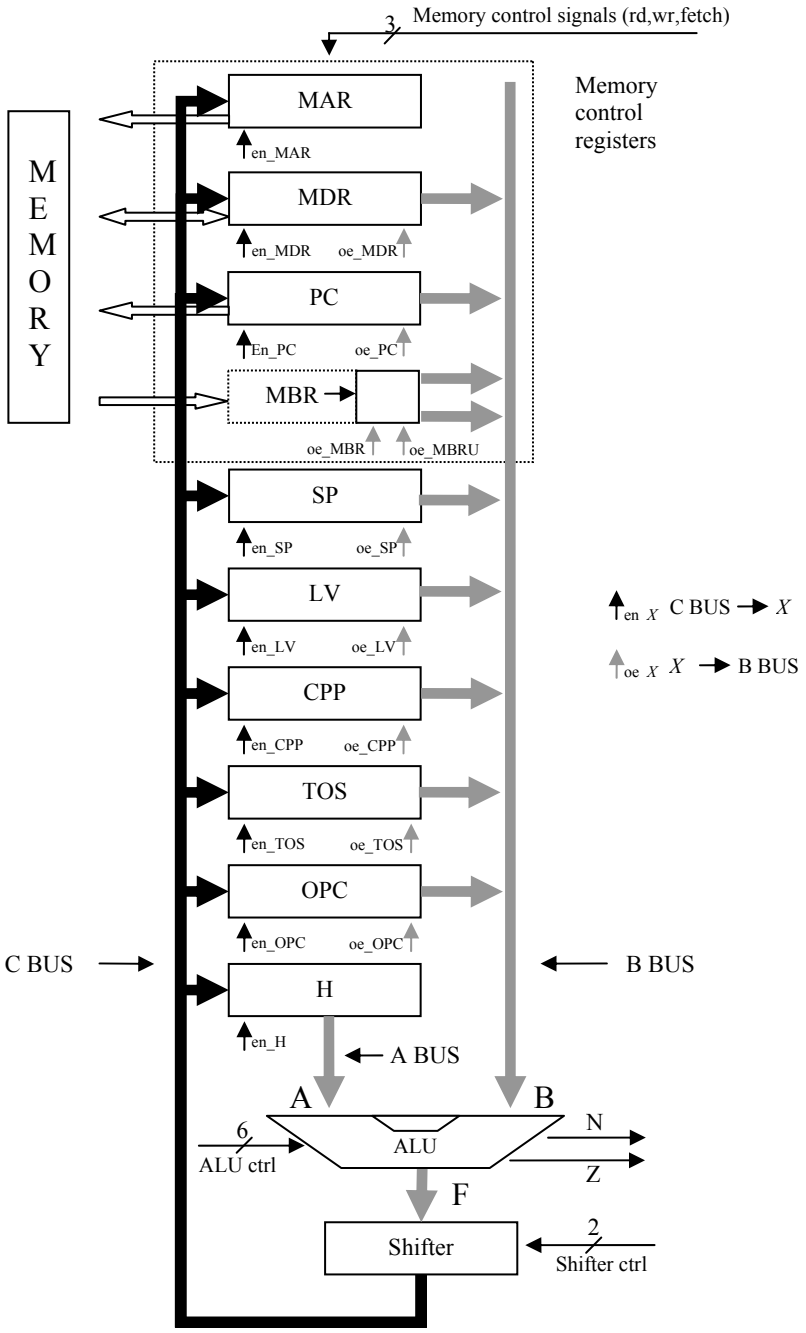

Figura 1 – Data Path da microarquitectura

O segundo grupo de registos é constituído por registos que partilham o mesmo conjunto de ligações, nomeadamente todos possuem ligação com os barramentos **C** (*input*) e **B** (*output*).

O único elemento do último grupo, o registo **H**, possui ligação com o barramento **C** (*input*) e o barramento **A** (*output*), sendo assim um registo com acesso priviligiado à **ALU**.

É de notar a presença de um ou dois sinais de controlo por baixo de cada um dos registos. Um sinal com o prefixo **en** activa a transferência entre o barramento **C** e o registo. Um sinal com o prefixo **oe** activa a transferência entre o registo e o barramento **B**. Como este barramento pode ser escrito por todos os registos que a ele estão ligados apenas um dos sinais **oe_xx** pode ser activado em cada ciclo para que não ocorram contenções nesse barramento. A transferência entre o registo **H** e a **ALU** está sempre activa.

A **ALU** implementa um conjunto de 16 operações apresentadas na Tabela 1, sendo a selecção da operação determinada pelo conjunto de sinais **ALU_ctrl**. Os *inputs* da **ALU** são designados **A** e **B** e possuem respectivamente ligação com os barramentos **A** e **B**. Existem ainda duas *flags* **N** e **Z** as quais sinalizam respectivamente se o resultado da operação seleccionada é negativo, segundo a convenção complemento para 2 (bit mais significativo igual a 1), ou zero.

F0	F1	ENA	ENB	INVA	INC	F(A,B)
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	NOT A
1	0	1	1	0	0	NOT B
1	1	1	1	0	0	A+B
1	1	1	1	0	1	A+B+1
1	1	1	0	0	1	A+1
1	1	0	1	0	1	B+1
1	1	1	1	1	1	B-A
1	1	0	1	1	0	B-1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	-1

Tabela 1-ALU

O **Shifter** possui ligação com a **ALU** (*input*), barramento **C** (*output*) e implementa um conjunto de três operações apresentado na Tabela 2. A operação é seleccionada pelo conjunto de sinais **Shifter_ctrl**.

SLL8	SRA1	F(X)
0	0	X
1	0	X<<8
0	1	X>>1,mantém bit de sinal

Tabela 2 –Shifter

Sincronização do Data Path

Uma vez apresentados cada um dos elementos constituintes do **Data Path** interessa analisar o que ocorre durante um ciclo operação. Como se pode constatar na Figura 2, cada ciclo de operação possui relação com o sinal de *clock*: possui a mesma frequência e tem início no correspondente flanco descendente. No primeiro ciclo

encontram-se representados quatro intervalos de tempo correspondentes ás várias fases de operação, designados subciclos. Assim a sequência de subciclos é seguinte:

1. Configuração dos sinais de controlo (Δw)
2. Transferência de registos para os barramentos **A** e **B** (Δx)
3. Operação da **ALU** e **Shifer** (Δy)
4. Propagação dos resultados ao barramento **C** (Δz)

Cada um destes subciclos é implicitamente determinado pelo atraso associado ao conjunto de componentes envolvidos em cada operação. Cada transferência de informação produz um resultado o qual só pode ser considerado estável após um intervalo de tempo. Por exemplo, no caso da **ALU** os inputs só podem ser considerados estáveis a partir do instante Δw+Δx, e o *output* após o intervalo Δw+Δx+Δy.

Ao contrário dos subciclos, que são implícitos, existem dois eventos que se assumem grande importância no ciclo do **Data Path**: o flanco descendente que inicia o ciclo; o flanco ascendente que desencadeia a carga de registos a partir do barramento **C** e memória.

O funcionamento correcto da microarquitectura é assim fortemente condicionado pela característica do sinal de *clock* (frequência, *duty cycle*), que deve satisfazer os requisitos temporais mostrados na Figura 2.

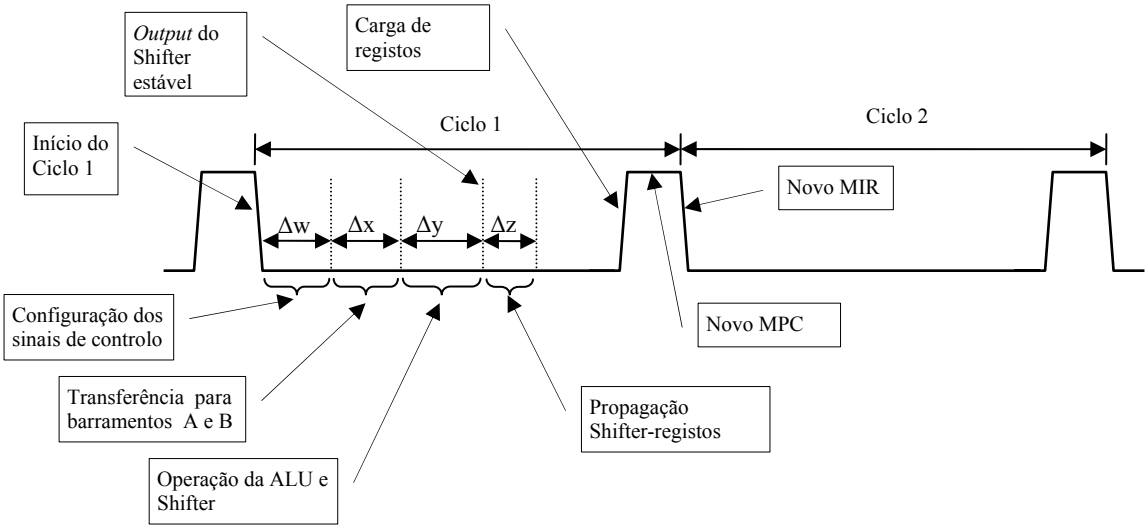

Figura 2 – Ciclo do Data Path

Interface com a memória

A microarquitectura possui duas formas de interface com a memória: um porto que permite o endereçamento de palavras de 32 bits e um porto que permite o endereçamento de palavras de 8 bits.
O primeiro porto é controlado pelo par de registos **MAR** (*Memory Address Register*) e **MDR** (*Memory Data Register*).
O segundo porto é controlado pelo par de registos **PC** (*Program Counter*) e **MBR** (*Memory Byte Register*).

Os dois pares de registos referem-se a duas partes da memória semanticamente distintas: O par de registos **MAR/MDR** é utilizado para ler e escrever na memória de dados, e o par **PC/MBR** é utilizado para ler da memória o código **ISA** executável.

O registo **MDR** é utilizado para ler ou escrever palavras de 32 bits endereçadas pelo registo **MAR** e o registo **MBR** é utilizado para ler palavras de 8 bits endereçadas pelo registo **PC**. No entanto o registo **MBR** possui a particularidade de ser o único registo de 8 bits, e existe uma ligação entre este registo e o barramento **B** (de 32 bits). Então existe a necessidade de transformar um valor de 8 em 32 bits. Neste sentido existem dois sinais que determinam a forma como este registo é transferido para o barramento **B**: **oe_MBRU** e **oe_MBR**. O sinal **oe_MBRU** activa a transferência do valor do registo para os 8 bits menos significativos do barramento **B**, sendo preenchidos os 24 bits mais significativos de **B** com o valor zero. O sinal **oe_MBR** activa uma transferência designada por "extensão de sinal" na qual é igualmente transferido o valor do registo para os oito bits menos significativos de **B** mas os 24 bits mais significativos passam a possuir o valor do bit mais significativo de **MBR**, dando lugar a um valor de 32 bits com sinal.

Associado ao grupo de registos de controlo da memória existe ainda um conjunto de três sinais de controlo: **rd**, **wr** e **fetch**. Os dois primeiros estão associados ao par de registos **MAR/MDR** e activam respectivamente as operações de leitura e escrita. O sinal **fetch** está associado ao par de registos **PC/MBR** e activa a operação de leitura.

Na Figura 1 pode ser identificado o conjunto de sinais que participam no controlo do **Data Path**:

- 9 sinais que controlam a transferência entre o barramento **C** e os registos.
- 9 sinais que controlam a transferência entre os registos e o barramento **B**.
- 8 sinais que controlam a operação da **ALU** e **Shifter**
- 2 sinais que controlam a operação na memória via **MAR/MDR**
- 1 sinal que controla a operação na memória via **PC/MBR**

Este conjunto de sinais a operação a realizar pelo **Data Path** durante um ciclo, por exemplo: qual o registo que escreve no barramento **B**, que operação a realizar na **ALU** e quais os registos a serem carregados com o resultado da **ALU**.

Relativamente às operações na memória o caso é algo diferente: uma operação de leitura ou escrita desencadeada num ciclo **k** não é concluída durante o mesmo. No mínimo a operação só pode ser desencadeada após os registos de endereços terem sido actualizados (**MAR** ou **PC**). No caso de uma operação de escrita existe outro registo a ter em conta, o registo que contém o valor transferir (**MDR**).

A operação inicia-se no flanco ascendente do ciclo **k** após os registos terem sido actualizados. A operação decorre durante o ciclo **k+1** e termina no flanco ascendente do mesmo ciclo, na mesma altura em que ocorre a transferência entre o barramento **C** e os registos. O resultado da operação só pode ser utilizado no ciclo **k+2**. Esta regra é muito importante como se constatará ao longo do capítulo.

2.1.3 Control Path

Como foi referido inicialmente o **Control Path** implementa o mecanismo que controla o **Data Path** permitindo a execução de um conjunto de instruções **ISA**. Nesta microarquitectura o controlo é implementado à custa da técnica de microprogramação.

Microinstruções

O **Data Path** pode ser inteiramente controlado por um conjunto de 29 sinais.
A Figura 3 representa o formato da microinstrução a qual é constituída pelos seguintes 6 campos: **Addr, JAM, ALU, C, Mem e B**

Os dois primeiros campos, **Addr** e **JAM**, codificam a próxima microinstrução. O campo **ALU** codifica o conjunto de sinais de controlo das unidades funcionais **ALU** e **Shifter**. O campo **C** codifica a transferência de registos a partir do barramento **C**. O campo **Mem** codifica o conjunto de sinais de controlo da memória. Por último, o campo **B** codifica a transferência de registos para o barramento **B**. Esta última transferência é peculiar pois, em cada ciclo só pode ocorrer a transferência de único registo para o barramento **B**. Consequentemente ao contrário dos outros campos nos quais existe uma correspondência directa entre cada um dos bits constituintes e o sinal correspondente, no campo **B** é utilizada outra codificação: 4 bits codificam 9 selecções possíveis (ver legenda da Figura 3).

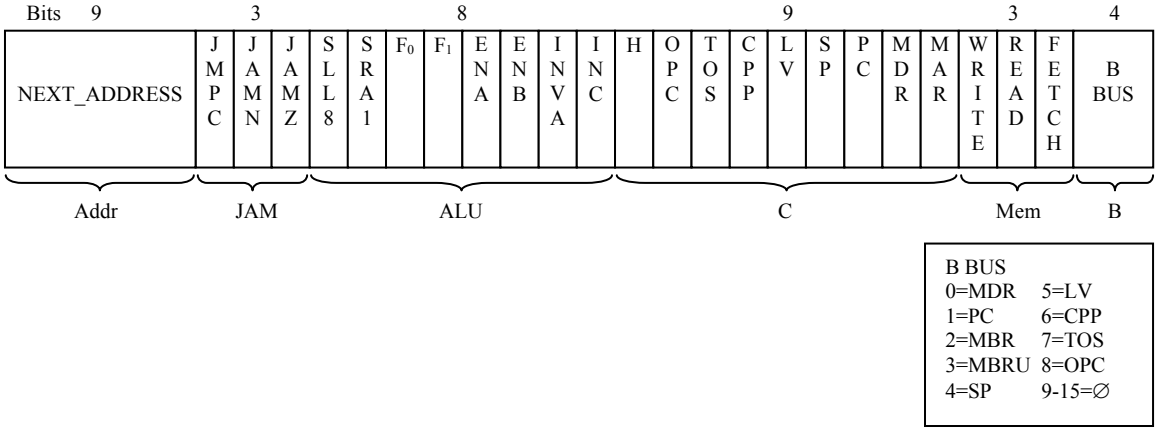

Figura 3 – Formato da microinstrução

Controlo de microinstruções: Mic1

O papel do **Control Path** consiste em determinar que sinais devem ser activados em cada ciclo de operação. Em cada ciclo são activados os sinais de controlo do **Data Path** e é seleccionada a próxima microinstrução.

As entidades envolvidas neste processo são o **Control Store** e os registos **MPC** (*MicroProgram Counter*) e **MIR** (*MicroInstruction Register*), e lógica de selecção que determina o endereço da próxima microinstrução.

O **Control Store** é uma memória que contém o conjunto de microinstruções que constituem o microprograma que implementa o interpretador de instruções **ISA**. Esta memória tem capacidade para 512 microinstruções de 36 bits.

O registo **MPC** contém o endereço da próxima microinstrução a ser lida do **Control Store** enquanto o registo **MIR** guarda a microinstrução corrente cujos bits estão directamente ligados aos sinais de controlo.

A Figura 4 apresenta o diagrama blocos da microarquitectura **Mic1**. À esquerda encontra-se o **Data Path** e à direita encontra-se o **Control Path**.

Com o **Control Path** surge um conjunto de novos componentes: um descodificador 4-16, os registos **MPC** e **MIR**, o **Control Store**, dois *flip-flops* e um bloco de lógica envolvido no cálculo do endereço da próxima microinstrução.

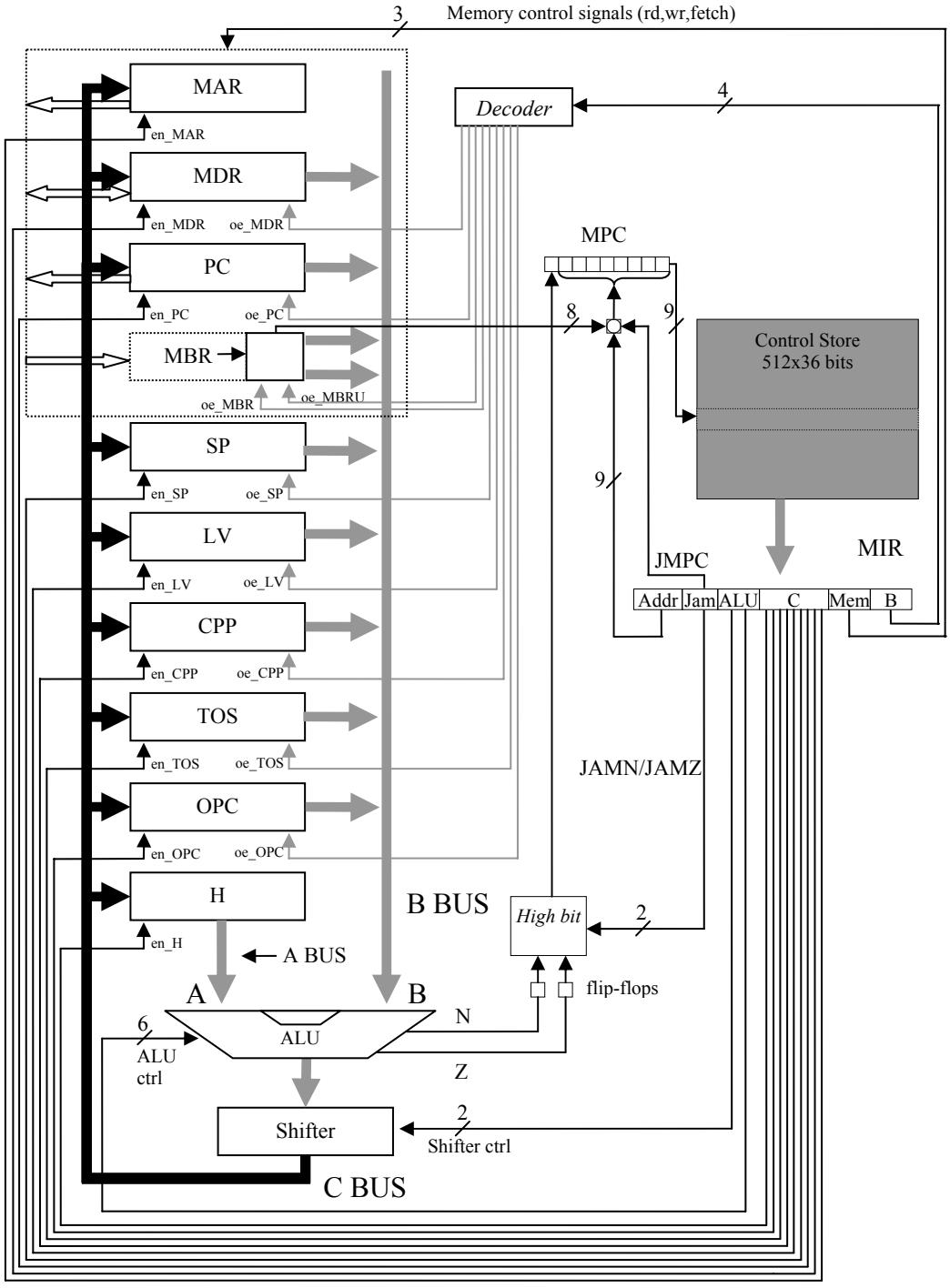

Figura 4 – Diagrama da microarquitectura

Funcionamento

O funcionamento do Mic1 pode ser resumido à seguinte sequência que é repetida interminavelmente:

1. O registo **MIR** é carregado a partir da microinstrução endereçada por **MPC** no **Control Store**. O registo **MIR** torna-se estável após o intervalo **Δw**.
2. Os sinais de controlo propagam-se do registo **MIR** ao **Data Path**. É transferido o conteúdo de um registo para o barramento **B**. São configuradas as operações nas unidades funcionais **ALU** e **Shifter**. Os *inputs* da **ALU** tornam-se estáveis decorrido o intervalo de tempo **Δw+Δx** .
3. As unidades funcionais **ALU** e **Shifter** entram em operação. O respectivo *output* estabiliza após o intervalo **Δw+Δx+Δy.**
4. O *output* do **Shifter** propaga-se ao longo do barramento **C** com destino aos registos e após o intervalo de tempo **Δw+Δx+Δy+Δz** encontra-se estável**.**
5. Carga dos registos, a partir do barramento **C** ou da memória, no flanco ascendente do ciclo de **clock**. O processo de cálculo do endereço da próxima microinstrução inicia-se nesta altura, tornando-se o valor de **MPC** estável um pouco antes do início do próximo ciclo do **Data Path**.

Processo de cálculo do endereço da nova microinstrução

Ao contrário do conjunto de instruções **ISA** que constitui o programa executável, o conjunto de microinstruções não são executadas ordenadamente pelo endereço que ocupam no **Control Store**. Nas microinstruções o ordenamento é explícito: cada uma destas codifica o endereço da seguinte, nos grupos de campos **Addr** e **JAM**. Este processo desencadeia-se da seguinte forma. Inicialmente o valor do campo **NEXT_ADDRESS** é copiado para o registo **MPC**. Na mesma altura, o grupo **JAM** é examinado e surgem dois cenários:

- **JAM=0**
  O campo **NEXT_ADDRESS** corresponde ao endereço da próxima microinstrução.
- **JAM≠0**
  O endereço da próxima instrução será função do valor de **NEXT_ADDRESS, JAMN**, **JAMZ**, **JMPC**, **MBR** e *flip-flops* **N** e **Z.**

Os bits **JAMN** e **JAMZ** contribuem para o valor do bit mais significativo de **MPC** da seguinte forma:

HighBit = (JAMN AND N) OR (JAMZ AND Z) OR NEXT_ADDRESS[8]

Na prática este esquema permite condicionar selecção da próxima microinstrução ao resultado da **ALU** (negativo ou zero).

O cálculo do valor de **MPC** termina com o bit **JMPC.** A contribuição de **JMPC** é fundamental uma vez que permite condicionar a execução do microprograma ao valor do registo **MBR**, atingindo-se assim a capacidade de executar código **ISA**. Assim quando **JMPC≠0** é obtido como uma função lógica que usa o valor presente no registo **MBR**, de acordo com a seguinte expressão:

MPC = {HighBit, NEXT_ADDRESS[7:0]} ⇐ **JMPC=0**
MPC = {HighBit, (NEXT_ADDRESS[7:0] OR MBR[7:0])} ⇐ **JMPC≠0**

2.2 Macroarquitectura: IJVM

Como foi referido no início do capítulo a macroarquitectura descreve os aspectos do microprocessador que são visíveis ao programador. A IJVM pelo facto de ser um subconjunto da JVM possui o mesmo tipo de macroarquitectura: é uma *stack machine*. Muito embora seja um subconjunto a IJVM sofre igualmente algumas simplificações. Algumas das simplificações situam-se a nível da semântica de evocação de métodos ou na organização da memória. Ao longo desta secção vão ser apresentadas as características mais importantes.

2.2.1 Modelo de Memória

A IJVM define quatro áreas de memória apresentadas na Figura 5.

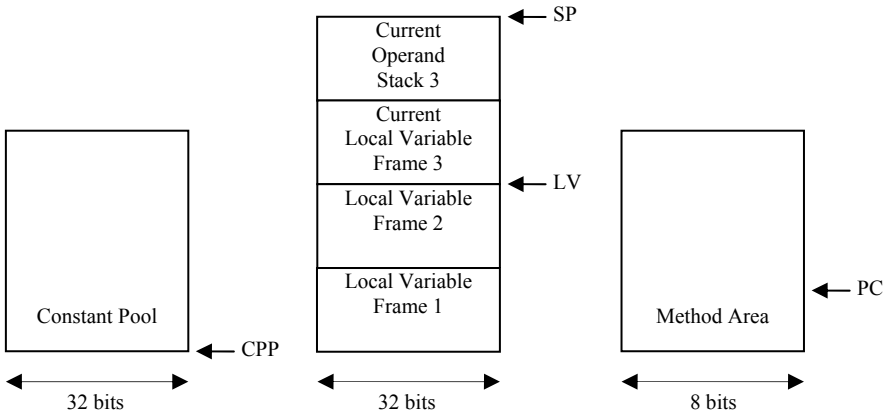

Figura 5 – Modelo de memória

Constant Pool
Zona de memória apenas de leitura destinada a guardar constantes. Esta área é carregada na mesma altura que o programa executável. O endereçamento a esta zona utiliza o registo **CPP** que indica a posição da primeira constante.

Local Variable Frame
Zona de memória na qual são guardadas as variáveis locais à evocação do método corrente. O endereçamento a esta zona é conseguido á custa do registo **LV** que aponta para a base.

Operand Stack
Zona de memória dinâmica utilizada para guardar os operandos e encontra-se situada imediatamente acima da **Local Variable Frame**. Pode ser endereçada à custa do registo **SP** que aponta para o topo.

Method Área
Zona de memória destinada a conter programa **IJVM** constituído pelo conjunto de métodos. O endereçamento nesta zona utiliza o registo **PC**.

Os registos **CPP**, **LV**, e **SP** endereçam palavras de 32 bits e **PC** que endereça palavras de 8 bits.

O conjunto de instruções não permite endereçar a memória directamente. Aliás o conjunto de registo é totalmente escondido.

Relativamente á organização da memória, como pode ser constatado na Figura 5, as zonas **Local Variable Frame** e a **Operand Stack** partilham uma mesma zona de memória. Ao longo desta zona de existem várias instancias de cada uma destas, dispostas contiguamente: duas por evocação de método.

2.2.2 Conjunto de instruções

A Tabela 3 apresenta o conjunto de instruções que constituem a **IJVM**. A primeira coluna apresenta o *opcode* da instrução (*operation code*), a segunda apresenta a mnemónica e a terceira dá uma breve descrição da instrução.

Opcode	Mnemónica	Descrição
0x00	NOP	Não faz nada
0x10	BIPUSH *byte:i8*	Transfere o valor *byte* para a *stack* sob a forma de inteiro com sinal
0x13	LDC_W *index:u16*	Transfere para a *stack* a palavra do **Contant Pool** indexada por *index*
0x59	DUP	Duplica o topo da *stack*
0x57	POP	Retira uma palavra da *stack*
0x5F	SWAP	Troca o topo da *stack* com a palavra imediatamente abaixo
0x60	IADD	Retira duas palavras da *stack* e coloca o resultado da soma
0x64	ISUB	Retira duas palavras da *stack* e coloca o resultado da diferença
0x7E	IAND	Retira duas palavras da *stack* e coloca o resultado da operação lógica AND
0x80	IOR	Retira duas palavras da *stack* e coloca o resultado da operação lógica OR
0xA7	GOTO *offset:i16*	Salto incondicional
0x99	IFEQ *offset:i16*	Retira uma palavra da *stack* e salta caso seja zero
0x9B	IFLT *offset:i16*	Retira uma palavra da *stack* e salta caso seja negativa
0x9F	IF_ICMPEQ *offset:i16*	Retira duas palavras da *stack* e salta se forem iguais
0x15	ILOAD *varnum:u8*	Coloca na *stack* o conteúdo da variável local *varnum*
0x36	ISTORE *varnum:u8*	Transfere o topo da *stack* para a variável local *varnum*
0x84	IINC *varnum:u8 const:i8*	Incrementa a variável local *varnum* em *const* unidades
0xC4	WIDE	Prefixo de alteração da próxima instrução
0xB6	INVOKEVIRTUAL *disp:u16*	Evocação do método
0xAC	IRETURN	Returno do método
0xFC	IN	Transfere a *stack* o valor disponível no porto de *input*
0xFD	OUT	Transfere para o porto de *output* o topo da *stack*
0xFF	HALT	Pára a execução do programa

Tabela 3 – Conjunto de instruções

Existem instruções com zero, um e dois operandos. Relativamente ao tipo dos operandos os prefixos *i/u* indicam se o operando é um inteiro com ou sem sinal, seguindo-se o número de bits.

A primeira instrução, **NOP**, é certamente a instrução mais usual na maioria das **ISAs** e pura e simplesmente "não faz nada" a não ser gastar tempo de processamento.

As instruções **BIPUSH**, **LDC_W** e **ILOAD** permitem carregar para a *stack* valores de fontes diversas nomeadamente do próprio operando da instrução, da **Constant Pool** e da **Local Variable Frame**.

Existem instruções que permitem efectuar operações aritméticas (**IAND** e **ISUB**) e lógicas (**IAND** e **IOR**), que utilizam como argumentos as duas palavras do topo da *stack* substituindo-as pelo resultado da respectiva operação.

O conjunto de instruções (**DUP**, **POP** e **SWAP**) permite manipular a *stack*. **DUP** duplica a palavra que está no topo da stack. POP retira uma palavra da *stack.* **SWAP** efectua uma permutação entre a palavra do topo da *stack* com a imediatamente abaixo.

Existem quatro instruções de controlo de fluxo (**GOTO**, **IFEQ**, **IFLT** e **IF_ICMPEQ**) que implementam saltos incondicionais (**GOTO**) e saltos condicionais (**IFEQ**, **IFLT** e **IF_ICMPEQ**). Todas elas possuem um único operando, um inteiro de 16 bits com sinal, que indica qual o deslocamento relativo ao endereço do *opcode* da instrução. As instruções **IFEQ** e **IFLT** utilizam como critério o valor do topo da *stack* (igual a zero ou negativo). A instrução **IF_ICMPEQ** utiliza como critério a comparação das duas palavras do topo da stack. Nestas últimas instruções após a execução da instrução o objecto do teste é retirado da *stack* (uma ou duas palavras).

As instruções **ILOAD** e **ISTORE** e **IINC** permitem manipular o conjunto de variáveis locais do método em execução. A instrução **ILOAD** já foi referida e transfere para o topo da *stack* a variável local indexada pelo operando *varnum* (8 bits). A instrução **ISTORE** faz precisamente o oposto: transfere do topo da *stack* uma palavra para a variável local indexada pelo operando *varnum*. Por último, **IINC** é uma única instrução composta por dois operandos: *varnum* e *const*, e permite, numa só instrução incrementar, o valor de uma variável indexada por *varnum* em *const* unidades. O operando *const* pode tomar valores entre -128 e +127. Esta última instrução permite ainda uma maior compactação do código além de facilitar a vida ao programador, já que é uma operação frequente. Relativamente a este último conjunto de instruções pode colocar-se um problema, pois só podem ser manipuladas as primeiras 256 variáveis locais devido ao operando *varnum* que permite apenas esta gama de referências. Este cenário verifica-se igualmente na **JVM**. Uma das formas que ultrapassar esta limitação consiste na inclusão de uma pseudo instrução (**WIDE**) que altera a versão da próxima instrução. No caso da **IJVM** a instrução **WIDE** é utilizada em conjunto com as instruções **ILOAD** e **ISTORE**: quando esta precede cada uma das instruções os operandos aumentam o domínio para versões de 16 bits, permitindo desta forma que as referidas instruções possam manipular 65536 em vez de 256 variáveis locais.

Existem instruções que não fazem parte da versão original da **IJVM** nomeadamente as instruções de **I/O** (**IN** e **OUT**) e a instrução **HALT**. As duas primeiras são importantes caso se pretenda que o programa **IJVM** interaja com o exterior. A instrução **IN** coloca no topo da *stack* o valor presente no porto de *input* e a instrução **OUT** envia para o porto de *output* o valor do topo da *stack*. A instrução **HALT** permite parar a execução do programa **IJVM**.

Por último, restam as instruções mais complexas que implementam a evocação e retorno de métodos: **INVOKEVIRTUAL** e **IRETURN**

A instrução **INVOKEVIRTUAL** permite evocar um método com base no operando *disp* e em parâmetros colocados na **Operand Stack**. O conjunto de parâmetros utilizados na evocação deve ser concordante, em número, com a definição do método. Na **IJVM** a restrição da compatibilidade de tipos não se coloca pois existe apenas um tipo. Relativamente ao operando *disp*, este endereça no **Constant Pool** uma palavra

que contém o endereço na **Method Area** da definição do método. A Figura 6 apresenta uma situação na qual estão representados dois métodos: **A** e **B**.

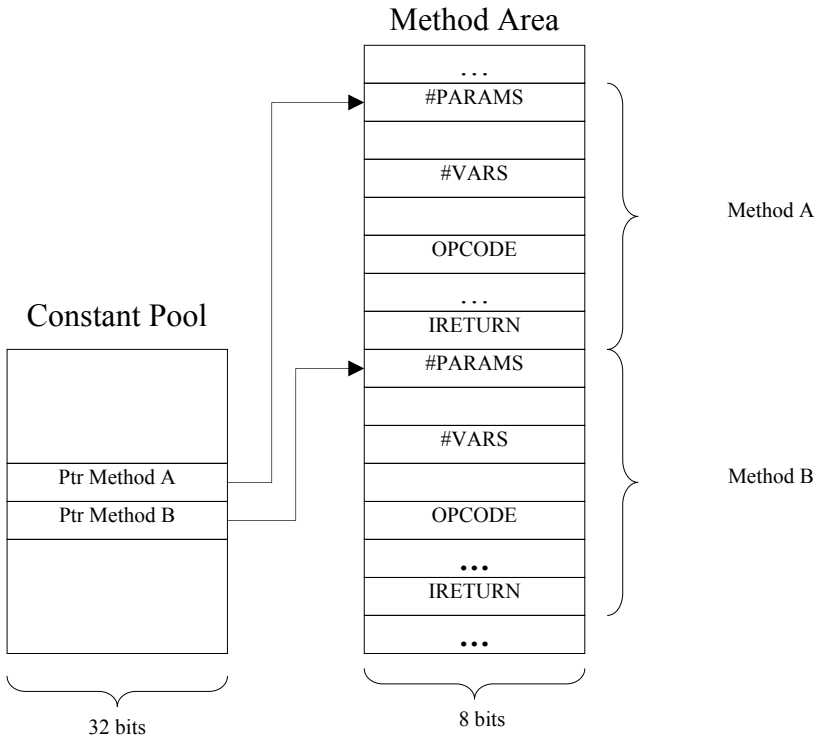

Figura 6 – Formato de um método

O formato do método consiste num cabeçalho constituído por quatro bytes seguido pelo respectivo conjunto de instruções. O cabeçalho contém informação relativa ao número de parâmetros formais (os primeiros dois bytes) e ao número de variáveis locais (os dois bytes seguintes). Esta informação será utilizada para estabelecer as novas **Local Variable Frame** e **Operand Stack**. Imediatamente a seguir ao cabeçalho (quinto byte) situa-se o conjunto de instruções que compõe o método. A última instrução deste conjunto é normalmente **IRETURN**.

O processo de evocação poderá consistir na seguinte sequência:

1. É colocado na *stack* um valor designado **OBJREF**, que é um parâmetro obrigatório e cujo valor não possui qualquer significado (será utilizado para guardar informação de ligação).
2. São também sucessivamente colocados na *stack* cada um dos parâmetros do método.
3. É executada a instrução **INVOKEVIRTUAL** indicando o respectivo operando a palavra do **Constant Pool** que contém o endereço do método.

Antes da execução da instrução **INVOKEVIRTUAL**, o registo **SP** aponta para o último parâmetro e o registo **LV** aponta para a base da **Local Variable Frame**.

Com base no registo **SP** e no número de parâmetros do método é determinada a posição ocupada pela palavra **OBJREF**; O registo **SP** é incrementado tendo em consideração o número de variáveis locais e duas novas palavras as quais serão utilizadas para guardar o valor actual do registo **LV** e o valor do registo **PC** alterado

de forma a apontar para o endereço da instrução que sucede **INVOKEVIRTUAL**. Uma vez registados os valores dos registos **LV** e **PC** respectivamente nas posições **SP** e **SP-1**, são estabelecidos novos valores para estes registos: o registo **LV** ficará a apontar para posição ocupada pelo parâmetro **OBJREF** e **PC** passa a apontar para o *opcode* da primeira instrução do método evocado (quinto byte). Finalmente na palavra endereçada agora por **LV** (**OBJREF**) é guardado o valor **SP-1**. Nesta nova configuração esta palavra passa a ser designada **LinkPtr**.

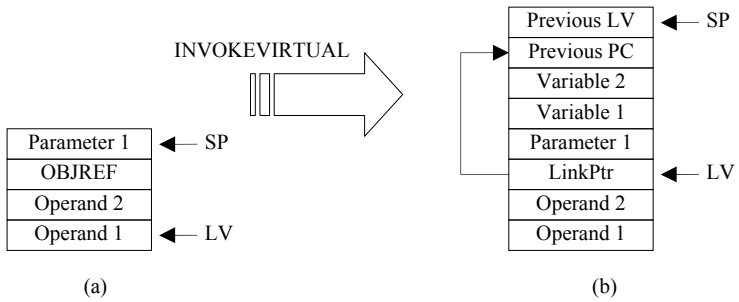

Figura 7 – Evocação de INVOKEVIRTUAL

A Figura 7 representa o conjunto de alterações na *stack* durante o processo de evocação de um método. Neste caso assume-se que o método evocado possui dois parâmetros (**OBJREF** é sempre incluído) e duas variáveis locais. Convém referir que antes da execução da instrução **INVOKEVIRTUAL** existem dois operandos na stack, os quais deixam de estar acessíveis no novo contexto. O estado inicial da *stack* na Figura 7 é peculiar já que é anterior a qualquer evocação de um método.

Por último a instrução **IRETURN** permite devolver o controlo de execução ao estado anterior à evocação do método e coloca o valor de retorno na **Operand Stack**. Utilizando o registo **LV** é obtido o valor de **LinkPtr**. O registo **SP** toma o valor de **LV** sendo posteriormente repostos os valores de **PC** e **LV** á custa das palavras endereçadas por **LinkPtr**. Por último resta transferir para a palavra endereçada por **SP** o valor de retorno do método que se encontrava guardado no registo **TOS**. A Figura 8 ilustra este processo para uma situação inversa da que foi apresentada na Figura 7.

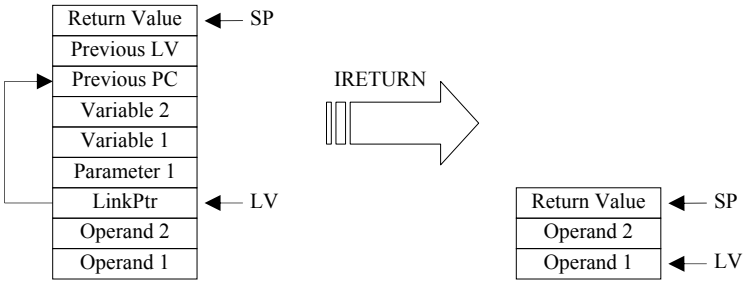

Figura 8 – Evocação de IRETURN

2.3 Implementação da IJVM no Mic1

2.3.1 Notacão

Uma vez definida a microarquitectura **Mic1** e a macroarquitectura **IJVM** resta a seguinte questão: de que forma pode a segunda ser implementada na primeira? A

resposta a esta questão passa pela definição de um microprograma que ao ser executado na microarquitectura seja capaz de interpretar código **IJVM**. Este microprograma será alojado no **Control Store** e terá a dimensão máxima de 512 microinstruções.

O microprograma consiste num conjunto de microinstruções podendo cada uma destas ser descrita numa palavra de 36 bits. Para microprogramas de alguma dimensão este tipo de codificação torna-se uma tarefa algo penosa e susceptível a erros.

Outra das características do microprograma é não possuir relação entre o ordenamento das microinstruções no **Control Store** e a ordem pela qual serão executadas. Cada microinstrução deve explicitamente codificar o endereço da próxima. No entanto existem microinstruções que devem ocupar posições específicas o que dificulta a tarefa da distribuição do conjunto de microinstruções pelo **Control Store**.

No sentido de facilitar a tarefa da escrita do microprograma surge então uma linguagem de programação designada **MAL** (*Micro Assembly Language*). Esta linguagem é uma notação simbólica que tem como orientação reflectir o conjunto de características da microarquitectura de forma a permitir um controlo mais apertado sobre o processo de geração do microcódigo.

A linguagem **MAL** possui dois tipos de entidades: as **directivas** que permitem impor restrições na geração do código e as **instruções** que permitem descrever cada uma das microinstruções.

Geralmente, a primeira parte da instrução tem o seguinte formato:

DEST_SPEC* “=“ *SOURCE_EXPR

DEST_SPEC* := *DEST* (“=” *DEST*)

***DEST* := N | Z | H | OPC | TOS | CPP | LV | SP | PC | MDR | MAR**

A definição ***DEST_SPEC*** suporta múltiplos elementos e indica o conjunto de registos activos do grupo **C**. Por exemplo no caso da instrução

H=OPC=TOS=CPP=LV=SP=PC=MDR=MAR=*SOURCE_EXPR*

são activados todos os bits do grupo **C**. A separação dos elementos pertencentes a ***DEST_SPEC*** ou a ***SOURCE_EXPR*** é feita á custa do último sinal de igualdade.

Convém referir que os registos **N** e **Z** são registos virtuais e não se traduzem na activação de qualquer elemento do grupo **C.** Consequentemente caso sejam estes os únicos elementos de ***DEST_SPEC*** não será activado qualquer elemento neste grupo.

SOURCE_EXPR define o conjunto de expressões admissíveis:

SOURCE_EXPR := ALU_EXPR SHIFTER_EXPR

ALU_EXPR **:= H |** ***SOURCE*** **| NOT H | NOT** ***SOURCE*** **| H "+"** ***SOURCE*** **| H "+"** ***SOURCE*** **"+" 1 | H "+" 1 |** ***SOURCE*** **"+" 1 |** ***SOURCE*** **"–" H |** ***SOURCE*** **"-" 1 | "-" H | H AND** ***SOURCE*** **| H OR** ***SOURCE*** **| 0 | 1 | -1**

SOURCE **:= MDR | PC | MBR | MBRU | SP | LV | CPP | TOS | OPC**

SHIFTER_EXPR **:= ">>" 1 | "<<" 8 | ∅**

ALU_EXPR e ***SHITER_EXPR*** definem o conjunto de elementos do grupo **ALU** e ***SOURCE*** define o registo activo no grupo **B**.

Não foram ainda contemplados os bits do grupo **Mem**: **rd, wr e fetch**. Para activar estes bits basta enumera-los utilizando as palavras reservadas **rd**, **wr** e **fetch**. Por exemplo a instrução

PC = PC + 1; fetch; wr

é traduzida numa microinstrução na qual se encontram activos os bits **fetch** e **rd**. É de notar que existe a necessidade de separar os elementos por ";".

Até agora foi estabelecida a relação entre os elementos da instrução e os grupos da microinstrução que descrevem a operação do **Data Path**. Nada foi ainda referido relativamente aos grupos **Addr** e **JAM**, os quais permitem encadear o conjunto de microinstruções.

Normalmente o compilador atribui um endereço a cada uma das instruções e configura os grupos **Addr** e **JAM** de forma que instruções escritas em linhas consecutivas sejam executadas consecutivamente. Existe no entanto a necessidade de controlar a execução quer seja de forma condicionada ou não condicionada.

Neste sentido surge uma nova entidade: o identificador da instrução. Cada instrução pode conter um identificador que pode ser usado como referência. Aliás a notação permite que existam instruções constituídas apenas pelo identificador (neste caso a microinstrução correspondente teria todos os grupos com excepção de **Addr** a zero).

Relativamente ao controlo não condicionado é utilizada a construção:

goto ***label***

na qual ***label*** representa o identificador da próxima instrução. Neste caso o grupo **Addr** será configurado de forma a apontar para a instrução referenciada e o grupo **JAM** terá o valor zero.

Relativamente ao controlo condicionado existem as seguintes primitivas:

1. **if (Z) goto L1 ; else L2** activa o bit **JAMZ**
2. **if (N) goto L1 ; else L2** activa o bit **JAMN**
3. **goto (MBR OR value), goto(MBR)** activam o bit **JMPC**

Nas duas primeiras a próxima microinstrução é condicionada pelos valores das *flags* **Z** e **N. L1 e L2** representam os identificadores das microinstruções candidatas, as quais serão colocadas no **Control Store** em endereços que partilham os 8 bits menos significativos. Neste caso **Addr** toma o valor dos oito bits menos significativos de **L1** ou **L2**.

Na última a próxima microinstrução é condicionada pelo resultado da expressão **MBR OR *value***, no qual **Addr** toma o resultado da expressão. Na forma **goto(MBR) Addr** toma o valor zero. Assim existem 256 sucessores possíveis para esta microinstrução.

Foram descritos os aspectos mais relevantes da notação **MAL** que serão úteis para a percepção do microprograma que será apresentado em seguida. Em [6] encontra-se o documento da especificação da linguagem suportada pelo compilador utilizado.

2.3.2 Microprograma

É chegada á altura de apresentar o microprograma que implementa o interpretador da **IJVM**. O microprograma utilizado encontra-se em anexo.

Convém fixar um conjunto de pressupostos utilizados nesta implementação. Um dos aspectos importantes é o papel desempenhado por cada um dos registos.

Os registos **MAR** e **MDR** são utilizados na interface à memória na qual estão definidas as zonas **Local Variable Frame**, **Operand Stack** e **Constant Pool**. Cada uma destas zonas possui um registo associado: o registo **LV** aponta para a base da **Local Variable Frame**; o registo **SP** aponta para o topo da **Operand Stack**; o registo **CPP** aponta para a base da **Constant Pool**.

Os registos **PC e MBR** são utilizados na interface à memória que contém a **Method Área**. O registo **PC** é utilizado para endereçar uma sequência de bytes que constitui o programa **IJVM,** e **MBR** é utilizado para guardar o *opcode* ou parte do operando.

O registo **H** devido á sua natureza funciona como acumulador.

Na maioria dos casos os registos **TOS** e **OPC** podem ser utilizados como registos temporários quando necessário. No caso do registo **TOS** existe uma situação excepcional: no início e o no fim de cada instrução deve guardar o elemento que se encontra no topo da *stack*. O registo **OPC** não possui papel determinado. É simplesmente um registo temporário.

Existe outro ponto que interessa recordar: o resultado de uma operação na memória iniciada numa microinstrução ***k*** só poderá ser utilizado na microinstrução ***k+2.*** Por exemplo se uma microinstrução activar o sinal ***fetch***, o resultado dessa operação só estará disponível em **MBR** após a execução da seguinte.

Relativamente á estrutura do microprograma destacam-se dois tipos de elementos:

1. Uma microinstrução designada **Main1**.
2. Rotinas de tratamento das instruções.

A microinstrução **Main1** desencadeia o seguinte conjunto de operações: incrementa o registo **PC**, ficando este a apontar para o primeiro byte que sucede o *opcode*; inicia

uma operação de leitura neste último endereço (**PC+1**); salta para a microinstrução que ocupa no **Control Store** um endereço igual ao valor actual do registo **MBR**.

Na microinstrução **Main1** parte do pressuposto que o registo **PC** foi previamente carregado com um endereço de uma posição de memória que contém um *opcode* e que este se encontra disponível no registo **MBR**.

Geralmente uma rotina de tratamento consiste numa sequência de microinstruções com características particulares. A primeira microinstrução da sequência ocupa no **Control Store** uma posição determinada pelo valor numérico do *opcode* da respectiva instrução e a última salta geralmente para **Main1**. Assume-se que na primeira microinstrução o registo **PC** aponta para o byte que sucede o *opcode* o qual estará disponível em **MBR** na microinstrução seguinte. Este valor pode corresponder a um novo *opcode*, estar relacionado com um operando ou já não pertencer ao programa (o que acontece na última instrução). Caso uma instrução não possua operandos a respectiva rotina de tratamento não utiliza o referido byte e consequentemente não altera o valor de **PC**.

É de notar que muito embora **Main1** possa ser a primeira instrução do microprograma não ocupará a primeira posição no **Control Store**. Esta posição está por exemplo reservada para a primeira e única microinstrução da rotina de tratamento da instrução **NOP** que possui o *opcode* 0x00.

Todos os endereços do **Control Store** correspondentes a *opcodes* são efectivamente reservados. Todas as microinstruções que não as que iniciam sequências, são distribuídas pelas posições que se encontram vazias.

O microprograma pode então ser descrito como um ciclo interminável no qual **Main1** salta para a rotina de tratamento da instrução relacionada com **MBR** e esta, quando termina, salta para **Main1**.

Por último interessa definir como começa o processo, ou seja o que acontece quando o sistema é ligado. Para isso é neccessário definir a configuração inicial dos registos envolvidos: os registos **MPC**, **PC** e **MBR**. Para **MPC**=0x0, **PC**=0xFFFFFFFF e **MBR**=0x0 (*opcode* da operação NOP) a sequência de microintruções é a seguinte **nop1**; **Main1**; **nop1**; **Main1**; rotina de tratamento do *opcode* da 1ª instrução, Main1,… até á última instrução, normalmente **halt1**. Nesta configuração, e para o microprograma apresentado, a rotina de tratamento da primeira instrução só começa a ser executada após quatro microinstruções.

3 Implementação em FPGA

3.1 Alterações á proposta de Tanenbaum

No capítulo anterior foram apresentadas os aspectos mais relevantes da microarquitectura **Mic1**, da macroarquitectura **IJVM** e da implementação em microcódigo, que estabelece a relação entre as duas primeiros.

Se considerarmos apenas a microarquitectura, com a excepção da semântica da memória, é relativamente fácil implementar o conjunto em *hardware* reconfigurável, pois está bem descrita em [1]. Cada um dos componentes tem regras bem definidas o que facilita a implementação utilizando por exemplo uma linguagem de descrição de hardware. Em termos de implementação este foi um dos primeiros passos no projecto: descrever o microprocessador sem contar com a memória.

O problema reside quando se pretende implementar o sistema completo: microprocessador + memória, pois restam alguns detalhes importantes por definir ou mesmo adaptar, de forma a tornar a implementação possível. Estas adaptações devem no entanto permitir manter conjunto de características importantes do sistema. Seguidamente será detalhado o conjunto de alterações efectuadas.

Um desses aspectos é a interface à memória externa, apresentada na Figura 9. A interface descrita é demasiado simplificada, consistindo apenas em três sinais de controlo, **read**, **write** e **fetch**, além de dois pares de barramentos, respectivamente de endereços e dados.

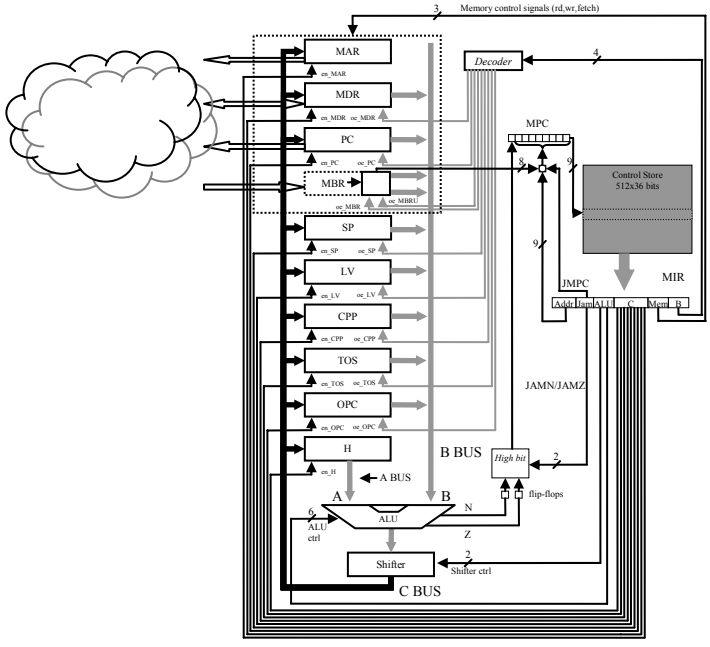

Figura 9 – Diagrama da microarquitecura original

A versão original apenas sugere um esquema de endereçamento para o caso de uma memória endereçável ao byte e como permitir passar a vê-la de duas formas diferentes: como um conjunto de palavras de 8 bits ou como um conjunto de palavras de 32 bits. O esquema sugerido é simples. Caso seja endereçada uma palavra de 8 bits, o endereço não sofre qualquer transformação. Caso seja endereçada uma palavra de 32 bits, o endereço é multiplicado por 4. Assim sendo o registo **PC** permite

endereçar 4.294.967.296 palavras de 8 bits e **MAR** permite endereçar 1.073.741.824 palavras de 32 bits.

Ainda relativamente à memória, a microarquitectura impõe a restrição de acesso concorrente. Neste caso, o ideal seria possuir duas memórias independentes às quais estariam respectivamente ligados os registos **MAR/MDR** e **PC/MBR**.

Por último, interessa definir qual é a quantidade de memória mínima que o sistema deve possuir. Após considerar algumas alternativas definiu-se a seguinte:

Utilizar memória externa à **FPGA** e criar um componente de interface a esta de forma a isolar o tipo de solução.

Decidiu-se alterar o *data width* de 32 para 16 bits e fixar os requisitos da quantidade de memória para 64KB + 2x64KB. A relação entre os novos valores do *data width* e o tamanho da memória deve-se a uma questão de coerência, já que assim o espaço de endereçamento permitido pelos registos é utilizador na totalidade ou seja, alterando os barramentos de endereçamento (ligados a **PC** e **MAR**) de 32 para 16 bits, estes passam a endereçar 64K elementos. Pelas mesmas razões todos os outros registos do **Data Path**, com excepção de **MBR**, sofrem esta alteração.

Em resumo foram efectuadas as seguintes alterações à proposta original do **Mic1** e **IJVM**, descritas no capítulo anterior:

1. *O data width* da microarquitectura passa a 16 bits.
2. Espaço de endereçamento da memória de programa passa a 64Kx8 bits.
3. Espaço de endereçamento da memória de dados passa a 64Kx16 bits

3.2 Plataforma de Desenvolvimento

Uma vez reajustada a microarquitectura, a implementação foi condicionada pela plataforma de desenvolvimento que foi entretanto adoptada. Convém então fazer uma breve descrição da mesma, sendo apresentadas algumas das características que poderão ter condicionado a implementação.

A plataforma utilizada é basicamente composta por dois componentes:
1. A plataforma de desenvolvimento para **FPGA**.
2. Um módulo adicional.

A plataforma de desenvolvimento, apresentada na Figura 10, é um *evaluation board* de suporte a desenvolvimento de projectos para a família **Spartan-IIE** da **Xilinx** desenvolvida pela Avnet. A **FPGA** (**XC2S200E**, *package* **FT256**) possui uma capacidade equivalente a 200K *gates*. O oscilador utilizado na plataforma é de 50MHz.

Devido ao custo e conjunto de características que possui, a plataforma torna-se interessante como suporte ao desenvolvimento de projectos de sistemas digitais de alguma dimensão.

A **FPGA**, pode ser configurada por *download* através da interface **JTAG**, ou então o *download* pode ficar a cargo da **PROM** que equipa a mesma *board*. A configuração da **PROM** é igualmente efectuada através da interface **JTAG**. A **PROM** permite, no

arranque do sistema, evitar a utilização de um PC, um cabo **JTAG** e *software* de *download.*

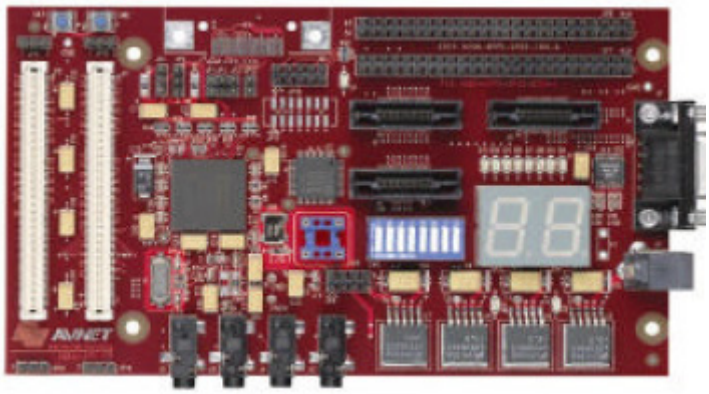

Figura 10 – Plataforma base

Em termos de *I/O* disponível na *board*, existem cerca de 100 sinais que ligam a **FPGA** aos conectores de expansão: quatro pares de conectores diferenciais ligados ao conector **LVDS** (**MDR**). As restantes linhas de *I/O*, encontram-se ligadas a oito *LEDs*, a oito *DIP-switches*, a dois *push-buttons*, a um *transceiver* **RS232** e a um *Codec* (**I2C** e **I2S**).

Uma vez que a plataforma base não apresenta uma solução para o problema da memória externa, foi necessário utilizar o módulo adicional, apresentado na Figura 11, designado ***Communication/Memory Module***, desenvolvido também pela **Avnet**. Este módulo permite estender a plataforma, dotando-a de recursos de memória e comunicações. O módulo encontra-se ligado à plataforma base através do barramento **AvBus**.

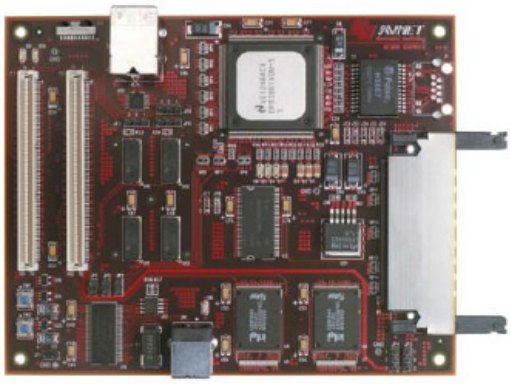

Figura 11 – Módulo adicional

O conjunto de recursos disponibilizado é o seguinte:

- **SDRAM** – 64 MB
- **FLASH** – 16 MB
- **SRAM** – 1MB
- Transceiver **Ethernet** de 10/100/1000 Mbps
- Interface **USB** 2.0 – conector tipo B
- *Transceiver* **IrDA** – compatível IrDA 1.1
- Interface **PC CARD** – compatível **PCMCIA** tipo 1 e 2
- Conectores de *I/O* **AvBus**

A interface da **FPGA** com módulo é feita via o barramento **AvBus** o qual inclui barramentos de endereços e dados, de 32 bits, e um conjunto de sinais de controlo dos vários componentes.

O módulo interpõe *buffers* na interface entre os barramentos de endereços e dados, e os vários integrados de memória, com a excepção, da **SDRAM** e **PCCard** que se encontram directamente ligadas. Este facto pode ser constatado no diagrama de blocos da Figura 12.

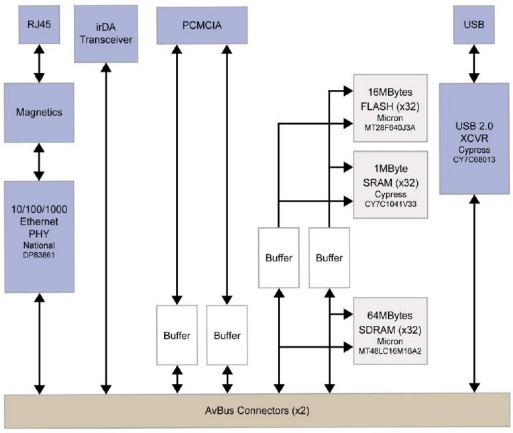

Figura 12 – Diagrama de blocos do modulo adicional

Convém no entanto detalhar um pouco mais cada um dos recursos de memória que o módulo disponibiliza:

SDRAM
64MB organizados em 16Mx32. Os dois integrados de SDRAM, de 256Mbits, encontram-se directamente ligados ao barramento **AvBus**.

FLASH
16 MB organizados em 4Mx32. Os dois integrados da **FLASH** encontram-se ligados aos *buffers* de endereços e dados. É de notar que esta memória pode ser programa via **JTAG**, utilizando para isso ferramentas de programação **JTAG**.

SRAM
1MB organizados em 256Kx32. Composta por dois integrados, cuja organização é 256Kx16, e encontram-se também ligados aos *buffers* de endereços e dados.

Perante este cenário foi decidido utilizar a **SRAM** como dispositivo de memória externa. A escolha prendeu-se essencialmente com razões de simplicidade de interface e performance.

Infelizmente apesar de ser constituída por dois integrados, estes partilham o barramento de endereços, o que invalida a hipótese de serem utilizados como memórias independentes.

3.3 MemCtrl

Acima foi brevemente referido um componente cujo papel seria servir de interface entre a microarquitectura e a memória, permitindo tornar o integrado da **RAM**

compatível com o conjunto de sinais e comportamento especificados. Este componente passa a ser designado **MemCtrl**.

Na microarquitectura é especificada que a interface com a **RAM** é feita á custa de dois pares de barramentos aos quais estão ligados os registos **MAR/MDR** e **PC/MBR**, que por sua vez permitem o acesso a duas memórias: de dados e programa. São também especificados três sinais de controlo: **read**; **write**; **fetch**. Os sinais **read** e **write** activam respectivamente as operações de leitura e escrita na memória de dados. O sinal **fetch**, activa a operação de leitura na memória de programa.

É também especificado que o resultado de uma operação na memória, desencadeada num microciclo **k**, só estará disponível no final do microciclo **k+1**. O valor resultante só poderá ser utilizado no microciclo **k+2**. É ainda referido que podem ser desencadeadas duas operações de memória desde que em memórias distintas.

Como existe apenas uma só memória, a solução é o **MemCtrl** multiplexar o acesso, no tempo, e utilizar uma gama diferente de endereços para a memória de programa e dados.

Este deve ser capaz em apenas um ciclo do **Data Path**, efectuar ambas as operações na memória. Aliás, pode ser precisamente o **MemCtrl** a gerar o sinal de *clock* da microarquitectura.

Esse sinal de *clock* deverá ser suficiente para o **MemCtrl** ter tempo para executar sequencialmente: operação na memória de programa; operação na memória de dados.

Após alguns testes, com a memória, optou-se pela seguinte configuração: o **MemCtrl** funciona á frequência base, 50MHz e gera um sinal de *clock* com uma frequência de 5MHz e *duty cycle* de 33.3%, que será usado na microarquitectura. Consequentemente, cada microciclo passa a consumir 200ns.

A Figura 13 apresenta um diagrama temporal no qual se estabelece a relação ente os sinais de *clock* do **MemCtrl** e da microarquitectura. No diagrama são assinalados dois instantes importantes: o instante no qual ocorre a transferência do resultado da operação de memória para os registos e o instante no qual são registados os sinais de controlo.

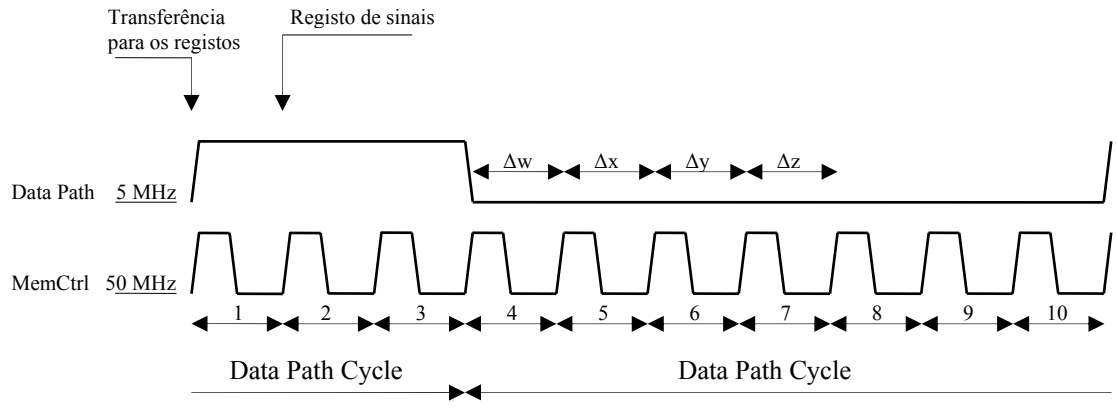

Figura 13 – Novo ciclo do Data Path

Por registo de um sinal entende-se fixar o valor do mesmo em determinado evento (tipicamente num flanco do sinal de *clock*). No **MemCtrl**, o conteúdo nos barramentos, de endereços e dados, e os respectivos sinais de controlo, são transferidos para registos internos, no instante assinalado no diagrama, e mantém-se

até novo registo.

Analisando o diagrama apresentado na Figura 14, constata-se que existem dois pares de sinais: **inst_op/r_inst_op** e **data_op/r_data_op** correspondendo a respectivamente operações nas memórias de programa **(_inst**) e dados **(_data**). Cada par tem a forma de **x/y**, em que **x** corresponde ao sinal vindo da microarquitectura, e **y** corresponde ao sinal registado no **MemCtrl**. O mesmo acontece com os valores dos respectivos barramentos que ligam a microarquitectura ao **MemCtrl**. Esta aproximação permite impor ordem no acesso à memória evitando alguns problemas caso ocorra algum desfasamento entre os sinais; permite uma certa margem de manobra.

No **MemCtrl** os sinais são então registados 20ns após o evento de flanco ascendente do sinal de *clock* de 5MHz (início do ciclo 2). Nesta altura todos os registos ligados ao barramentos de endereços e dados, e respectivos sinais, encontram-se estáveis e em condições ideais para serem registados. Os sinais registados são disponibilizados pelo **MemCtrl** para a microarquitectura de forma a permitir sincronizar a interface ao mesmo.

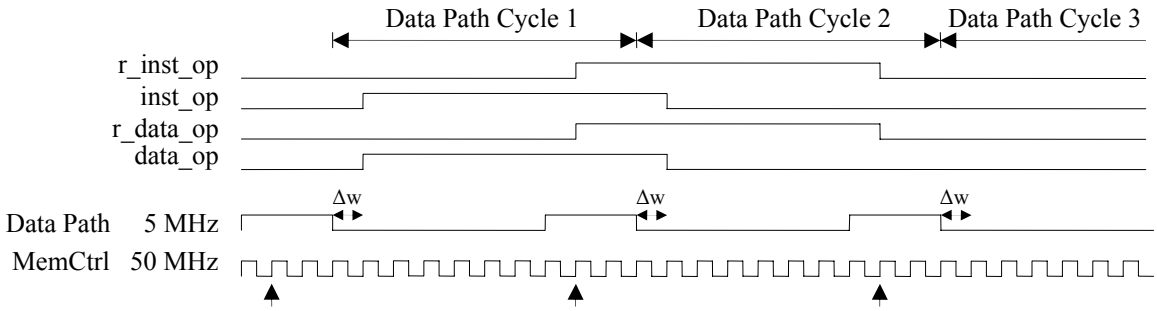

Figura 14 – Diagrama de operação na memória

Além da implementar a interface á memória o **MemCtrl** está também envolvido na interface ao *I/O*. A opção foi utilizar um endereço relativo à memória de dados e o respectivo conjunto de sinais. Assim quando é desencadeada operação de leitura ou escrita na referida memória no endereço 0xFFFF não são desencadeadas as referidas operações na **SRAM** mas nas linhas de *I/O*.

Relativamente ao **MemCtrl**, apresentado na Figura 15, interessa então reter o papel que este desempenha nomeadamente:

1. Interface a **SRAM**, e indirectamente ao *I/O*
2. Registo dos sinais de controlo da memória
3. Geração do sinal de *clock* da microarquitectura

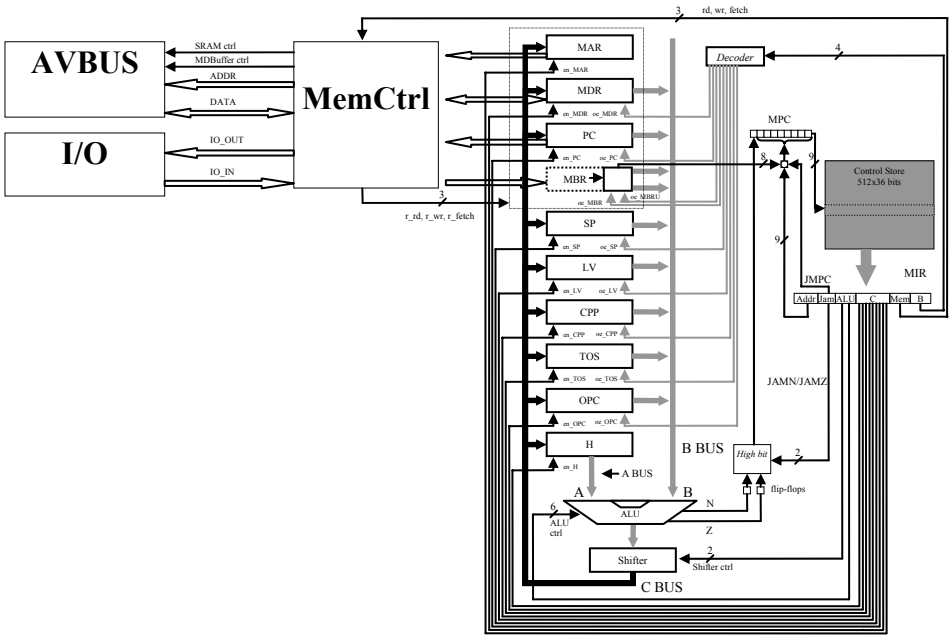

Figura 15 – Novo diagrama da microarquitecura

Os componentes que foram até a esta altura definidos, microprocessador e memória externa, já permitem executar um programa **IJVM**. Resta ainda um problema por resolver: o *download* do programa para memória. De forma a resolver este problema, foi desenvolvido mais um componente do sistema. Este componente foi designado **Monitor** e será em seguida brevemente apresentado.

3.4 Monitor

O Monitor começou por ser desenvolvido com o objectivo de criar um mecanismo de interface entre o PC e o sistema em desenvolvimento na **FPGA**. Neste sentido decidiu-se utilizar o *transceiver* **RS232** disponível na plataforma e implementar uma **UART**. A comunicação com o exterior seria assegurada por este componente, especialmente durante a fase de desenvolvimento.

É de notar que plataforma possui outros recursos de comunicação (**Ethernet**,**USB**,**IrDA**), mas o desenvolvimento de controladores para estes não se justificavam devido á complexidade. Pretendia-se somente condicionar o sistema na **FPGA** com base em sequências de caracteres enviados via porta **RS232** de um **PC**.

A primeira tarefa a resolver consistiu em implementar uma **UART**. A implementação em **Verilog** foi baseada na **UART** incluída no *demo* que acompanhava a plataforma de desenvolvimento (em **VHDL**). Convém notar que foi utilizada parte: o controlador de recepção designado **UARTrx**. O **UARTrx**, quando sintetizada na **FPGA**, implementa um circuito que trata da recepção de um carácter e notifica a chegada deste. Os parâmetros de comunicação **RS232** (*baudrate*,*start bits*, *stop bits*, *flow control*, *parity*) encontram-se predefinidos no componente. Nesta implementação o *baudrate* pode ser alterado, sendo os restantes parâmetros fixos, com os seguintes valores: *start bit*:1; *stop bits*:1; *flow control*:*none*; *parity*:*none*

O **Monitor** implementa uma máquina de estados que permite receber os caracteres, interpretar comandos e desencadear determinada acção.

Inicialmente foi utilizado para testar acesso á **SRAM**, mesmo antes de ter sido desenvolvido o **MemCtrl**. Nessa altura, a máquina de estados desencadeava leituras

ou escritas na memória, controlando totalmente o acesso à **SRAM**.

Numa última fase, foi incluído o **MemCtrl** no **Monitor** actual, mesmo antes de testá-lo com o microprocessador. A estratégia foi testar separadamente cada um dos componentes do sistema (sempre que possível), e posteriormente agregá-los e testar o conjunto.

A máquina de estados do Monitor permite interpretar os seguintes comandos:

rXXXX	leitura na memória de programa no endereço XXXX
wXXXXYY	escrita na memória de programa do valor YY no endereço XXXX
RXXXX	leitura na memória de dados no endereço XXXX
WXXXXYYYY	escrita na memória de dados do valor YYYY no endereço XXXX
tX	teste de funcionamento

Relativamente ao *output* do sistema, foi utilizado o conjunto de 8 *LEDs* e um *push button*, que dependendo do estado permite exibir a parte menos ou mais significativa do *output* (palavra de 16 bits).

Com o desenvolvimento do **Monitor** adquiriu-se a capacidade de ler e escrever na **SRAM**.

Optou-se por separar os dois circuitos, ou seja, a **FPGA** é configurada com um dos circuitos, microprocessador ou **Monitor**, dependendo do que se pretende. Desde que não seja necessário desligar a *board*, o conteúdo da **SRAM** mantém-se e permite a seguinte sequência:

1. Configura-se a **FPGA** como **Monitor** e escreve-se o programa na **RAM**
2. Configura-se a **FPGA** como microprocessador de forma a executar o programa
3. Opcionalmente, configura-se a novamente a **FPGA** como **Monitor** e examina-se o conteúdo da **RAM**

3.5 Implementação em Verilog

Em termos de implementação adoptou-se **Verilog** como linguagem de descrição de hardware. Os motivos da escolha prenderam-se essencialmente com o facto de ser mais intuitiva (face à alternativa VHDL), facilitando o processo de aprendizagem. Relativamente ao ambiente de desenvolvimento, foi utilizado o **ISE 6.0** da **Xilinx**.

O sistema desenvolvido encontra-se organizado em duas grandes entidades: a descrição do microprocessador – módulo **IJVM** apresentado na Figura 16, e a descrição do **Monitor** – módulo **IJVM_mon** apresentado na Figura 19.

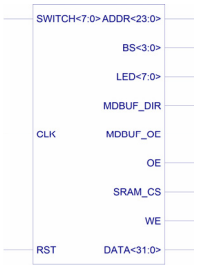

Figura 16 – módulo IJVM

Internamente o módulo **IJVM** é definido à custa de duas instâncias de componentes, apresentadas na Figura 17: uma instância do módulo **mic1** designada *mic1_1*, na qual estão descritas as entidades **Data Path** e **Control Path**; uma instância do módulo **mem_ctrl** designada *mem_ctrl_1* que descreve o **MemCtrl**.

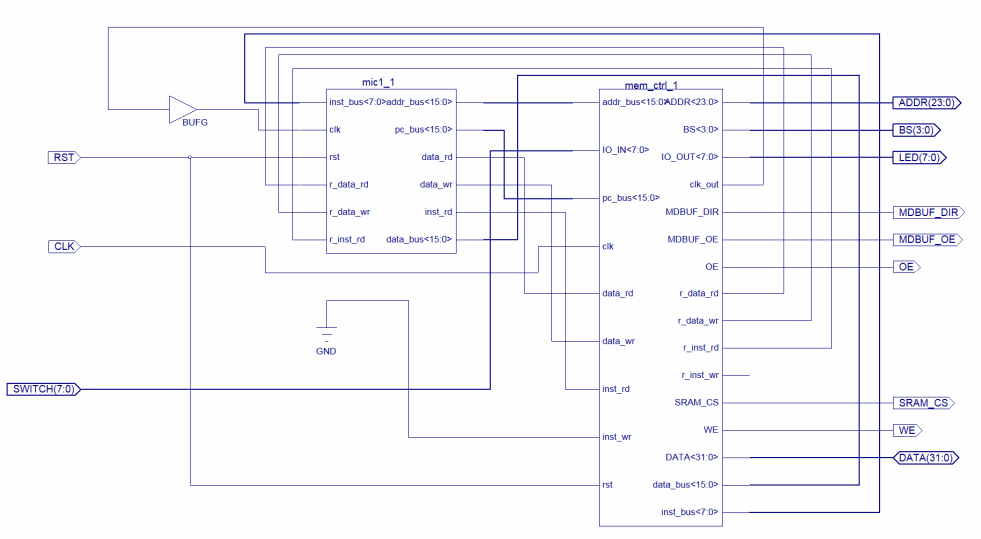

Figura 17 – Detalhe do módulo IJVM

A Figura 18 apresenta a estrutura interna do módulo **mic1** acima instanciado, na qual podem ser referenciadas duas instâncias: uma instância do módulo **datapath** designada *datapath1* e uma instância do módulo **controlpath** designada *controlpath1*. Convém o referir que internamente o módulo **controlpath** instancia um outro componente importante, o **Control Store**, definido no módudo **ROM**.

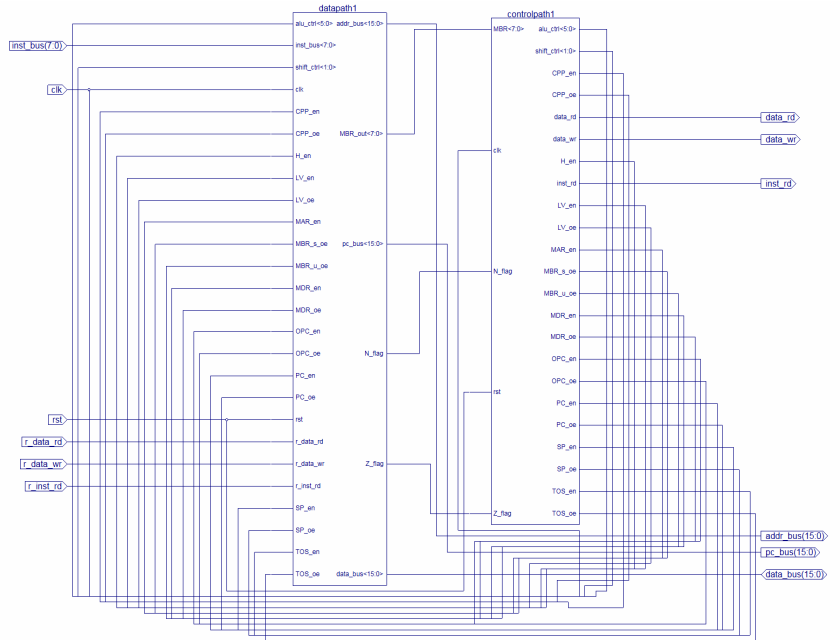

Figura 18 – Instância mic1_1

O segundo componente do sistema o **Monitor** é apresentado na Figura 19, e detalhado na Figura 20.

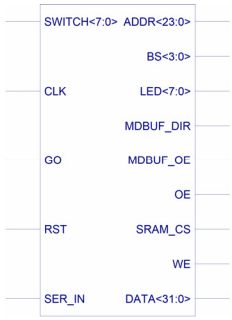

Figura 19 – Módulo IJVM_mon

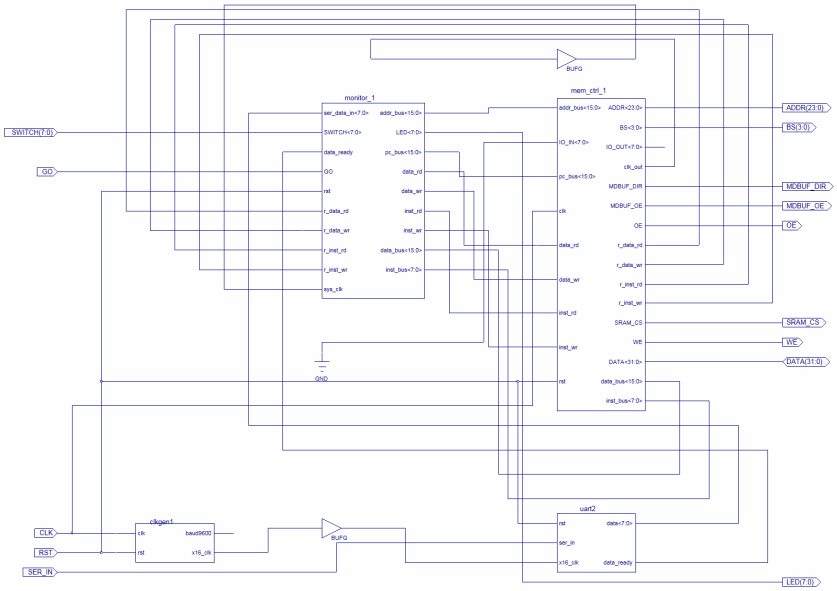

Figura 20 – Detalhe do módulo IJVM_mon

O componente é definido à custa das seguintes instâncias: uma instância do módulo **monitor** designada *monitor1*, na qual está descrita a implementação da máquina de estados que permite interpretar o conjunto de comandos de leitura e escrita na **RAM**; uma instância do módulo **mem_ctrl** designada *mem_ctrl_1*, descrita anteriormente; uma instância do módulo **uart_rx** designada *uart2* que implementa a parte de recepção do controlador **UART**; uma instância do módulo **uart_clkgen** designada *uart2* a qual define um gerador de sinal de *clock* utilizado no módulo **uart_rx**.

Em anexo encontra-se o código **Verilog** relativo à implementação do sistema. É constituído pelos seguintes ficheiros:

IJVM.v
mic1.v
mem_ctrl.v
controlpath.v;
ROM.v
datapath.v
IJVM_mon.v
monitor.v
uart_clkgen.v
uart_rx.v

Para além da descrição dos módulos, são acrescentados ao mesmo anexo dois ficheiros **IJVM.ucf** e **IJVM_mon.ucf**, nos quais é estabelecida a relação entre cada porto do componente e o correspondente na **FPGA**.

3.6 Demo

Como forma de compreender o sistema desenvolvido será interessante percorrer o processo, utilizando para isso um exemplo.

Vamos supor que se pretende implementar um programa que permita calcular a seguinte função:

$$f(x)=\begin{cases} x + f(x-1) & \Leftarrow x\neq 0 \\ 0 & \Leftarrow x=0 \end{cases}$$

Uma implementação do algoritmo em **IJVM** resultaria no seguinte programa:

MAIN	
10 00	BIPUSH 0
FD	OUT
10 00	BIPUSH 0
FC	IN
B6 00 00	INVOKEVIRTUAL *F*
FD	OUT
FF	HALT

F	
00 02 00 00	***PARMS:2 VARS:0***
15 01	ILOAD 1
59	DUP
99 00 0E	IFEQ *#ZERO*
59	DUP
10 00	BIPUSH 0
5F	SWAP
B6 00 00	INVOKEVIRTUAL *#F_REF*
60	IADD
AC	*ZERO*: IRETURN

A primeira coluna representa a codificação respectiva instrução, apresentada na coluna seguinte. Na conversão um dos aspectos mais sensíveis é a tradução das instruções de salto, apesar de todos eles serem relativos ao endereço do *opcode*.

Convém relembrar que o formato de um método inclui um cabeçalho que contém informação sobre o número de parâmetros e o número de variáveis locais (**OBJREF** conta como parâmetro).

Falta atribuir um endereço absoluto ao método ***F***. A primeira sequência é obrigatoriamente localizada no início da **Method Area**.

Caso o método ***F*** seja colocado na memória imediatamente a seguir ao primeiro bloco, este será colocado a partir do endereço 0x000B, o qual deve ser registado na primeira palavra do **Constant Pool**.

Assim na inicialização devem ser carregados para as respectivas memórias, o programa e o conjunto de constantes. O **Constant Pool** tem início no endereço 0xFF00 da memória de dados.

O próximo passo é configurar a **FPGA** com o **Monitor**, utilizando a ferramenta do ambiente de desenvolvimento designada **Impact**. Uma vez configurado o sistema pode ser utilizada uma aplicação que permita estabelecer uma ligação série configurada convenientemente (*baudrate: 115200, data bits: 8 bits, parity: none, stop bits:1 e flow control:none*) e utilizar o conjunto de comandos definidos. Por exemplo a sequência:

```
w000010 w000100 w0002FD w000310 w000400 w0005FC w0006B6 w000700 w000800 w0009FD w000aFF
```

permite escrever os 11 primeiros bytes do programa, e

```
WFF00000B
```

permite configurar a primeira palavra no **Constant Pool** de forma a apontar o endereço do método ***F***.

Depois de enviados para a memória o programa e o conjunto de constantes, a **FPGA** pode ser configurada como microprocessador, utilizando uma vez mais o **Impact**. Uma vez concluído o *download*, o microprocessador entra em funcionamento e o programa inicia a execução. Neste caso é calculado o resultado função f(x), sendo x o valor seleccionado nos *dip-switches* e o resultado (os 8 bits menos significativos) é apresentado no conjunto de **LEDs**.

4 Conclusão

Os objectivos do projecto foram atingidos uma vez que foi implementado com sucesso o microprocessador proposto, utilizando uma plataforma de desenvolvimento baseada em **FPGA.**

A validação do sistema foi efectuada à custa de pequenos programas sendo posteriormente utilizado o **Monitor** para *debug***.**

Teria sido interessante fazer interface a sensores e actuadores, permitindo por exemplo utilizar o sistema desenvolvido num *microrobot*.

Relativamente à microarquitectura implementada (***Mic1***), Tanenbaum propõe uma série de evoluções: **Mic2**, **Mic3** e **Mic4**. A última delas é quase equivalente, em termos de complexidade, a alguns dos microprocessadores actuais. Como evolução do sistema esta poderia ser uma das direcções a seguir.

Outra das áreas que seria interessante intervir seria o desenvolvimento de compiladores para esta arquitectura e ferramentas de suporte.

5 Bibliografia

1. Andrew S. Tanembaum, Structured Computer Organization, 4/e, Prentice Hall, 1998

2. Douglas J. Smith, HDL Chip Design, Doone Publications, 1997

3. J. Bhasker, Verilog HDL Synthesis, A Practical Primer, Star Galaxy Publishing, 1998

4. Herbert Taub, Digital Circuits and Microprocessors, McGraw-Hill, 1985

5. José Manuel Martins Ferreira, Introdução ao Projecto com Sistemas Digitais e Microcontroladores, FEUP edições

6. `Micro-Assembly Language (MAL) Specification`
http://www.ontko.com/mic1/mal.html

Índice de Figuras

Índice de Tabelas

Anexos

ijvm.mal

```
// note that this is nearly identical to the example
// given in Tanenbaum.  Note:
//
// 1) SlashSlash-style ("//") comment characters have been added.
//
// 2) "nop" has been added as a pseudo-instruction to indicate that
//    nothing should be done except goto the next instruction.  It
//    is a do-nothing sub-instruction that allows us to have MAL
//    statements without a label.
//
// 3) instructions are "anchored" to locations in the control
//    store as defined below with the ".label" pseudo-instruction
//
// 4) a default instruction may be specified using the ".default"
//    pseudo-instruction.  This instruction is placed in all
//    unused locations of the control store by the mic1 MAL assembler.
//

// labeled statements are "anchored" at the specified control store address
.label      nop1                0x00
.label      bipush1             0x10
.label      ldc_w1              0x13
.label      iload1              0x15
.label      wide_iload1         0x115
.label      istore1             0x36
.label      wide_istore1        0x136
.label      pop1                0x57
.label      dup1                0x59
.label      swap1               0x5F
.label      iadd1               0x60
.label      isub1               0x64
.label      iand1               0x7E
.label      iinc1               0x84
.label      ifeq1               0x99
.label      iflt1               0x9B
.label      if_icmpeq1          0x9F
.label      goto1               0xA7
.label      ireturn1            0xAC
.label      ior1                0xB0
.label      invokevirtual1      0xB6
.label      wide1               0xC4
.label      halt1               0xFF
.label      out1                0xFD
.label      in1                 0xFC

// default instruction to place in any unused addresses of the control store
.default    goto halt1

Main1       PC = PC + 1; fetch; goto (MBR)       // MBR holds opcode; get next byte; dispatch

nop1        goto Main1                                  // Do nothing

iadd1       MAR = SP = SP - 1; rd                       // Read in next-to-top word on stack
iadd2       H = TOS                                     // H = top of stack
iadd3       MDR = TOS = MDR + H; wr; goto Main1         // Add top two words; write to top of
stack

isub1       MAR = SP = SP - 1; rd                       // Read in next-to-top word on stack
isub2       H = TOS                                     // H = top of stack
isub3       MDR = TOS = MDR - H; wr; goto Main1         // Do subtraction; write to top of stack

iand1       MAR = SP = SP - 1; rd                       // Read in next-to-top word on stack
iand2       H = TOS                                     // H = top of stack
iand3       MDR = TOS = MDR AND H; wr; goto Main1       // Do AND; write to new top of stack

ior1        MAR = SP = SP - 1; rd                       // Read in next-to-top word on stack
ior2        H = TOS                                     // H = top of stack
ior3        MDR = TOS = MDR OR H; wr; goto Main1        // Do OR; write to new top of stack

dup1        MAR = SP = SP + 1                           // Increment SP and copy to MAR
dup2        MDR = TOS; wr; goto Main1                   // Write new stack word

pop1        MAR = SP = SP - 1; rd                       // Read in next-to-top word on stack
```

```
pop2                                        // Wait for new TOS to be read from memory
pop3        TOS = MDR; goto Main1           // Copy new word to TOS

swap1       MAR = SP - 1; rd                // Set MAR to SP - 1; read 2nd word from stack
swap2       MAR = SP                        // Set MAR to top word
swap3       H = MDR; wr                     // Save TOS in H; write 2nd word to top of stack
swap4       MDR = TOS                       // Copy old TOS to MDR
swap5       MAR = SP - 1; wr                // Set MAR to SP - 1; write as 2nd word on stack
swap6       TOS = H; goto Main1             // Update TOS

bipush1     SP = MAR = SP + 1               // MBR = the byte to push onto stack
bipush2     PC = PC + 1; fetch              // Increment PC, fetch next opcode
bipush3     MDR = TOS = MBR; wr; goto Main1 // Sign-extend constant and push on stack

iload1      H = LV                          // MBR contains index; copy LV to H
iload2      MAR = MBRU + H; rd              // MAR = address of local variable to push
iload3      MAR = SP = SP + 1               // SP points to new top of stack; prepare write
iload4      PC = PC + 1; fetch; wr          // Inc PC; get next opcode; write top of stack
iload5      TOS = MDR; goto Main1           // Update TOS

istore1     H = LV                          // MBR contains index; Copy LV to H
istore2     MAR = MBRU + H                  // MAR = address of local variable to store into
istore3     MDR = TOS; wr                   // Copy TOS to MDR; write word
istore4     SP = MAR = SP - 1; rd           // Read in next-to-top word on stack
istore5     PC = PC + 1; fetch              // Increment PC; fetch next opcode
istore6     TOS = MDR; goto Main1           // Update TOS

wide1       PC = PC + 1; fetch; goto (MBR OR 0x100)      // Multiway branch with high bit set

wide_iload1         PC = PC + 1; fetch            // MBR contains 1st index byte; fetch 2nd
wide_iload2         H = MBRU << 8                 // H = 1st index byte shifted left 8 bits
wide_iload3         H = MBRU OR H                 // H = 16-bit index of local variable
wide_iload4         MAR = LV + H; rd; goto iload3 // MAR = address of local variable to push

wide_istore1        PC = PC + 1; fetch            // MBR contains 1st index byte; fetch 2nd
wide_istore2        H = MBRU << 8                 // H = 1st index byte shifted left 8 bits
wide_istore3        H = MBRU OR H                 // H = 16-bit index of local variable
wide_istore4        MAR = LV + H; goto istore3    // MAR = address of local variable to store into

ldc_w1      PC = PC + 1; fetch              // MBR contains 1st index byte; fetch 2nd
ldc_w2      H = MBRU << 8                   // H = 1st index byte << 8
ldc_w3      H = MBRU OR H                   // H = 16-bit index into constant pool
ldc_w4      MAR = H + CPP; rd; goto iload3  // MAR = address of constant in pool

iinc1       H = LV                          // MBR contains index; Copy LV to H
iinc2       MAR = MBRU + H; rd              // Copy LV + index to MAR; Read variable
iinc3       PC = PC + 1; fetch              // Fetch constant
iinc4       H = MDR                         // Copy variable to H
iinc5       PC = PC + 1; fetch              // Fetch next opcode
iinc6       MDR = MBR + H; wr; goto Main1   // Put sum in MDR; update variable

goto1       OPC = PC - 1                    // Save address of opcode.
goto2       PC = PC + 1; fetch              // MBR = 1st byte of offset; fetch 2nd byte
goto3       H = MBR << 8                    // Shift and save signed first byte in H
goto4       H = MBRU OR H                   // H = 16-bit branch offset
goto5       PC = OPC + H; fetch             // Add offset to OPC
goto6       goto Main1                      // Wait for fetch of next opcode

iflt1       MAR = SP = SP - 1; rd           // Read in next-to-top word on stack
iflt2       OPC = TOS                       // Save TOS in OPC temporarily
iflt3       TOS = MDR                       // Put new top of stack in TOS
iflt4       N = OPC; if (N) goto T; else goto F  // Branch on N bit

ifeq1       MAR = SP = SP - 1; rd           // Read in next-to-top word of stack
ifeq2       OPC = TOS                       // Save TOS in OPC temporarily
ifeq3       TOS = MDR                       // Put new top of stack in TOS
ifeq4       Z = OPC; if (Z) goto T; else goto F  // Branch on Z bit

if_icmpeq1  MAR = SP = SP - 1; rd           // Read in next-to-top word of stack
if_icmpeq2  MAR = SP = SP - 1               // Set MAR to read in new top-of-stack
if_icmpeq3  H = MDR; rd                     // Copy second stack word to H
if_icmpeq4  OPC = TOS                       // Save TOS in OPC temporarily
if_icmpeq5  TOS = MDR                       // Put new top of stack in TOS
if_icmpeq6  Z = OPC - H; if (Z) goto T; else goto F      //If top 2 words equal,goto T,else goto F

T    OPC = PC - 1; fetch; goto goto2        // Same as goto1; needed for target address
```

```
F    PC = PC + 1                          // Skip first offset byte
F2   PC = PC + 1; fetch                   // PC now points to next opcode
F3   goto Main1                           // Wait for fetch of opcode

invokevirtual1      PC = PC + 1; fetch    // MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2      H = MBRU << 8         // Shift and save first byte in H
invokevirtual3      H = MBRU OR H         // H = offset of method pointer from CPP
invokevirtual4      MAR = CPP + H; rd     // Get pointer to method from CPP area
invokevirtual5      OPC = PC + 1          // Save Return PC in OPC temporarily
invokevirtual6      PC = MDR; fetch       // PC points to new method; get param count
invokevirtual7      PC = PC + 1; fetch    // Fetch 2nd byte of parameter count
invokevirtual8      H = MBRU << 8         // Shift and save first byte in H
invokevirtual9      H = MBRU OR H         // H = number of parameters
invokevirtual10     PC = PC + 1; fetch    // Fetch first byte of # locals
invokevirtual11     TOS = SP - H          // TOS = address of OBJREF - 1
invokevirtual12     TOS = MAR = TOS + 1   // TOS = address of OBJREF (new LV)
invokevirtual13     PC = PC + 1; fetch    // Fetch second byte of # locals
invokevirtual14     H = MBRU << 8         // Shift and save first byte in H
invokevirtual15     H = MBRU OR H         // H = # locals
invokevirtual16     MDR = SP + H + 1; wr  // Overwrite OBJREF with link pointer
invokevirtual17     MAR = SP = MDR;       // Set SP, MAR to location to hold old PC
invokevirtual18     MDR = OPC; wr         // Save old PC above the local variables
invokevirtual19     MAR = SP = SP + 1     // SP points to location to hold old LV
invokevirtual20     MDR = LV; wr          // Save old LV above saved PC
invokevirtual21     PC = PC + 1; fetch    // Fetch first opcode of new method.
invokevirtual22     LV = TOS; goto Main1  // Set LV to point to LV Frame

ireturn1    MAR = SP = LV; rd             // Reset SP, MAR to get link pointer
ireturn2                                  // Wait for read
ireturn3    LV = MAR = MDR; rd            // Set LV to link ptr; get old PC
ireturn4    MAR = LV + 1                  // Set MAR to read old LV
ireturn5    PC = MDR; rd; fetch           // Restore PC; fetch next opcode
ireturn6    MAR = SP                      // Set MAR to write TOS
ireturn7    LV = MDR                      // Restore LV
ireturn8    MDR = TOS; wr; goto Main1     // Save return value on original top of stack

halt1       goto halt1

out1        MAR = -1                      // compute OUT address
out2        MDR = TOS; wr                 // write to output
out3
out4        MAR = SP = SP - 1; rd         // decrement stack pointer
out5
out6        TOS = MDR; goto Main1

in1         MAR = -1; rd                  // compute IN address ; read from input
in2         MAR = SP = SP + 1             // increment SP; wait for read
in3         TOS = MDR; wr; goto Main1     // Write
```

IJVM.v

```
module IJVM
(
    CLK,
    RST,
    SWITCH, LED,
    ADDR, SRAM_CS, OE, WE, MDBUF_DIR, MDBUF_OE, BS, DATA
);

// CLOCK & RESET
input CLK;
input RST;

// IO
input [7:0] SWITCH;
output [7:0] LED;

// MEMORY SIGNALS
output [23:0] ADDR;
output SRAM_CS;
output OE;
output WE;
output MDBUF_DIR;
output MDBUF_OE;
output [3:0] BS;
inout [31:0] DATA;

wire sys_clk_w;
wire sys_clk;

wire r_data_rd;
wire r_data_wr;

wire r_inst_rd;
wire r_inst_wr;

wire data_rd;
wire data_wr;

wire inst_rd;
wire inst_wr;

wire [15:0] addr_bus;
wire [15:0] data_bus;
wire [15:0] pc_bus;
wire [7:0] inst_bus;

mem_ctrl mem_ctrl_1
(
    .clk(CLK),
    .rst(RST),
    .addr_bus(addr_bus), .data_bus(data_bus),
    .data_rd(data_rd), .data_wr(data_wr),
    .r_data_rd(r_data_rd), .r_data_wr(r_data_wr),
    .pc_bus(pc_bus), .inst_bus(inst_bus),
    .inst_rd(inst_rd), .inst_wr(inst_wr),
    .r_inst_rd(r_inst_rd), .r_inst_wr(r_inst_wr),
    .clk_out(sys_clk_w),
    .ADDR(ADDR), .SRAM_CS(SRAM_CS), .OE(OE), .WE(WE), .MDBUF_DIR(MDBUF_DIR), .MDBUF_OE(MDBUF_OE),
.BS(BS), .DATA(DATA),
    .IO_IN(SWITCH), .IO_OUT(LED)
);

BUFG buf_sys_clk ( .I(sys_clk_w), .O(sys_clk) );

mic1 mic1_1
(
    .clk(sys_clk),
    .rst(RST),
    .addr_bus(addr_bus),
    .data_bus(data_bus),
    .pc_bus(pc_bus),
    .inst_bus(inst_bus),
    .data_rd(data_rd), .data_wr(data_wr), .inst_rd(inst_rd),
```

```
    .r_data_rd(r_data_rd), .r_data_wr(r_data_wr), .r_inst_rd(r_inst_rd)
);

endmodule
```

mic1.v

```
module mic1
(
    clk,
    rst,
    addr_bus,
    data_bus,
    pc_bus,
    inst_bus,
    data_rd, data_wr, inst_rd,
    r_data_rd, r_data_wr, r_inst_rd
);
parameter pB=16;
parameter pL=8;

input clk;
input rst;

output [pB-1:0] addr_bus;
inout [pB-1:0] data_bus;
output [pB-1:0] pc_bus;
input [pL-1:0] inst_bus;

output data_rd, data_wr, inst_rd;
input r_data_rd, r_data_wr, r_inst_rd;

wire N_flag, Z_flag,
    MAR_en,
    MDR_en, MDR_oe,
    PC_en, PC_oe,
    MBR_u_oe, MBR_s_oe,
    SP_en, SP_oe,
    LV_en, LV_oe,
    CPP_en, CPP_oe,
    TOS_en, TOS_oe,
    OPC_en, OPC_oe,
    H_en;

wire[5:0] alu_ctrl;
wire [1:0] shift_ctrl;

wire [pL-1:0] wMBR;

datapath #(.pB(pB),.pL(pL)) datapath1(
    .clk(clk),
    .rst(rst),
    .addr_bus(addr_bus),
    .data_bus(data_bus),
    .pc_bus(pc_bus),
    .inst_bus(inst_bus),
    .MBR_out(wMBR),
    .N_flag(N_flag), .Z_flag(Z_flag),
    .MAR_en(MAR_en),
    .MDR_en(MDR_en), .MDR_oe(MDR_oe),
    .PC_en(PC_en), .PC_oe(PC_oe),
    .MBR_u_oe(MBR_u_oe), .MBR_s_oe(MBR_s_oe),
    .SP_en(SP_en), .SP_oe(SP_oe),
    .LV_en(LV_en), .LV_oe(LV_oe),
    .CPP_en(CPP_en), .CPP_oe(CPP_oe),
    .TOS_en(TOS_en), .TOS_oe(TOS_oe),
    .OPC_en(OPC_en), .OPC_oe(OPC_oe),
    .H_en(H_en),
    .r_data_rd(r_data_rd), .r_data_wr(r_data_wr), .r_inst_rd(r_inst_rd),
    .alu_ctrl(alu_ctrl), .shift_ctrl(shift_ctrl)
);

controlpath  #(.pB(pB),.pL(pL)) controlpath1(
    .clk(clk), .rst(rst),
    .MBR(wMBR),
    .N_flag(N_flag), .Z_flag(Z_flag),
    .MAR_en(MAR_en),
    .MDR_en(MDR_en), .MDR_oe(MDR_oe),
    .PC_en(PC_en), .PC_oe(PC_oe),
    .MBR_u_oe(MBR_u_oe), .MBR_s_oe(MBR_s_oe),
```

```
    .SP_en(SP_en), .SP_oe(SP_oe),
    .LV_en(LV_en), .LV_oe(LV_oe),
    .CPP_en(CPP_en), .CPP_oe(CPP_oe),
    .TOS_en(TOS_en), .TOS_oe(TOS_oe),
    .OPC_en(OPC_en), .OPC_oe(OPC_oe),
    .H_en(H_en),
    .alu_ctrl(alu_ctrl), .shift_ctrl(shift_ctrl),
    .data_rd(data_rd), .data_wr(data_wr), .inst_rd(inst_rd)
);

endmodule
```

datapath.v

```
module datapath
(
    clk,
    rst,

    addr_bus,
    data_bus,
    pc_bus,
    inst_bus,

    MBR_out,
    N_flag,
    Z_flag,
    MAR_en,
    MDR_en,
    MDR_oe,
    PC_en,
    PC_oe,
    MBR_u_oe,
    MBR_s_oe,
    SP_en,
    SP_oe,
    LV_en,
    LV_oe,
    CPP_en,
    CPP_oe,
    TOS_en,
    TOS_oe,
    OPC_en,
    OPC_oe,
    H_en,

    r_data_rd,
    r_data_wr,
    r_inst_rd,

    alu_ctrl,
    shift_ctrl
);

// defalt parameters, data width=16, pc data width=8
parameter pB=16;
parameter pL=8;

// clock & reset
input clk, rst;

// memory internal buses
output [pB-1:0] addr_bus;
inout [pB-1:0] data_bus;
output [pB-1:0] pc_bus;
input [pL-1:0] inst_bus;

output [pL-1:0] MBR_out;

// Z & N flip-flops, to control path
output N_flag, Z_flag;

// _en & _oe signals from control_path
input MAR_en,
    MDR_en, MDR_oe,
    PC_en, PC_oe,
    MBR_u_oe, MBR_s_oe,
    SP_en, SP_oe,
    LV_en, LV_oe,
    CPP_en, CPP_oe,
    TOS_en, TOS_oe,
    OPC_en, OPC_oe,
    H_en;

// registered mem ops
input r_data_rd, r_data_wr, r_inst_rd;
```

```
// ALU & Shifter ctrls
input [5:0] alu_ctrl;
input [1:0] shift_ctrl;

// bit registers
reg N_flag, Z_flag;

// X bit registers:
reg [pB-1:0]
    MAR,
    MDR,
    PC,
    SP,
    LV,
    CPP,
    TOS,
    OPC,
    H;

// pL bit registers:
reg [pL-1:0] MBR;

// internal datapath buses:
wire [pB-1:0] A, B, C;

// internal ALU buses
//
reg [pB-1:0] alu_output, shifter_output;

// address bus always has the contents of MAR:
assign addr_bus = MAR;

// PC bus always has the contents of PC:
assign pc_bus = PC;

// data_bus
assign data_bus = r_data_rd ? {pB{1'bz}} : MDR;

// MBR_out always has the contents of register MBR (to controlpath)
//
assign MBR_out = MBR;

// A bus always has the contents of register H:
//
assign A = H;

// registers to B bus
assign B = ( MDR_oe )   ? MDR : {pB{1'bz}};
assign B = ( PC_oe )    ? PC  : {pB{1'bz}};
assign B = ( SP_oe )    ? SP  : {pB{1'bz}};
assign B = ( LV_oe )    ? LV  : {pB{1'bz}};
assign B = ( CPP_oe )   ? CPP : {pB{1'bz}};
assign B = ( TOS_oe )   ? TOS : {pB{1'bz}};
assign B = ( OPC_oe )   ? OPC : {pB{1'bz}};

// MBR | MBRU -> B BUS:
//
assign B = ( MBR_s_oe ) ? { {pB-pL{MBR[pL-1]}}, MBR} : {pB{1'bz}};  // sign extend MBR
assign B = ( MBR_u_oe ) ? { {pB-pL{1'b0}}, MBR} : {pB{1'bz}};  // extend MBR with pB-pL zeroes

// C bus is driven by ALU output after the shifter
//
always @( A or B or alu_ctrl)
begin
    case ( alu_ctrl )
    6'b01_1000 : // A
           alu_output = A;
    6'b01_0100 : // B
           alu_output = B;
    6'b01_1010 : // NOT A
           alu_output = ~A;
    6'b10_1100 : // NOT B
           alu_output = ~B;
    6'b11_1100 : // A+B
           alu_output = A+B;
    6'b11_1101 : // A+B+1
           alu_output = A+B+1;
```

```
    6'b11_1001 : // A+1
            alu_output = A+1;
    6'b11_0101 : // B+1
            alu_output = B+1;
    6'b11_1111 : // B-A
            alu_output = B-A;
    6'b11_0110 : // B-1
            alu_output = B-1;
    6'b11_1011 : // -A (0-A)
            alu_output = -A;
    6'b00_1100 : // A and B
            alu_output = A & B;
    6'b01_1100 : // A or B
            alu_output = A | B;
    6'b01_0000 : // 0
            alu_output = 0;
    6'b01_0001 : // 1
            alu_output = 1;
    6'b01_0010 : // -1
            alu_output = -1;
    default : // set alu output to zeroes
            alu_output = 0;
    endcase
end

// Shifter: shifts alu_output
//
always @(shift_ctrl or alu_ctrl or alu_output)
begin
    case( shift_ctrl )
    2'b00 : // no shift operation
            shifter_output = alu_output;
    2'b01 : // shift right arithmetic
            shifter_output = {alu_output[pB-1], alu_output[pB-1:1]};
    2'b10 : // shift left one byte (8 bits)
            shifter_output = { alu_output[pB-pL-1:0], {pL{1'b0}} };
    default: // no shift operation
            shifter_output = alu_output;
    endcase
end

// C bus
//
assign C = shifter_output;

// datapath registers:
always @(posedge clk or posedge rst)
begin
    if ( rst ) begin
            MAR <= {pB{1'b0}};
            MDR <= {pB{1'b0}};
            PC <= {pB{1'b1}}; // 0xFFFF
            MBR <= {pB{1'b0}};
            SP <= {pB{1'b0}};
            LV <= {pB{1'b0}};
            CPP <= 16'hFF00; // 0xFF00
            TOS <= {pB{1'b0}};
            OPC <= {pB{1'b0}};
            H <= {pB{1'b0}};
            N_flag <= 1'b0;
            Z_flag <= 1'b0;
    end
    else    begin
            if ( MAR_en )
                    MAR <= C;

            // read from C bus has priority over read from external memory
            if ( MDR_en )
                    MDR <= C;
            else if ( r_data_rd )
                    MDR <= data_bus;

            if ( PC_en )
                    PC <= C;

            if ( r_inst_rd )
                    MBR <= inst_bus;
```

```
            if ( SP_en )
                    SP <= C;

            if ( LV_en )
                    LV <= C;

            if ( CPP_en )
                    CPP <= C;

            if ( TOS_en )
                    TOS <= C;

            if ( OPC_en )
                    OPC <= C;

            if ( H_en )
                    H <= C;

            // save shift_output result flags
            N_flag <= shifter_output[pB-1]; // shifter result (C bus) is negative
            Z_flag <= (shifter_output == 0);  // shifter result (C bus) is zero
    end
end

endmodule
```

controlpath.v

```
module controlpath
(
    clk, rst,
    MBR,
    N_flag, Z_flag,
    MAR_en,
    MDR_en, MDR_oe,
    PC_en, PC_oe,
    MBR_u_oe, MBR_s_oe,
    SP_en, SP_oe,
    LV_en, LV_oe,
    CPP_en, CPP_oe,
    TOS_en, TOS_oe,
    OPC_en, OPC_oe,
    H_en,
    alu_ctrl, shift_ctrl,
    data_rd, data_wr, inst_rd
);

parameter pB=16;
parameter pL=8;

input clk, rst;
input [pL-1:0] MBR;
input N_flag, Z_flag;

output MAR_en,
    MDR_en, MDR_oe,
    PC_en, PC_oe,
    MBR_u_oe, MBR_s_oe,
    SP_en, SP_oe,
    LV_en, LV_oe,
    CPP_en, CPP_oe,
    TOS_en, TOS_oe,
    OPC_en, OPC_oe,
    H_en;

output [5:0] alu_ctrl;
output [1:0] shift_ctrl;

output data_rd, data_wr, inst_rd;

reg [pL+27:0] MIR;
reg [pL:0] MPC;

wire [pL+27:0] wMIR;
ROM aROM(.a(MPC), .o(wMIR));

wire [pL:0] next_address;
wire JAMZ, JAMN, JMPC;
reg  high_bit;

//
// assign the MIR fields to the datapath control signals:
// BITS
//      3                   2                   1                       0
// 543210987  6 5 4  3 2 1 0 9 8 7 6  5 4 3 2 1 0 9 8 7  6 5 4  3 2 1 0
//
// 9          3      8                9                  3       4
// ---------  -----  ---------------  -----------------  -----  -------
// XXXXXXXXX  J J J  S S F F E E I I  H O T C L S P M M  W R F  B B B B
// NEXT_ADDR  M A A  L R 0 1 N N N N    P O P V P C V A  R E E  3 2 1 0
//            P M M  L A     A B V C    C S P       R R  I A C
//            C N Z  8 1         A                      T D T
//                                                      H
assign inst_rd = MIR[4],
       data_rd = MIR[5],
       data_wr = MIR[6],
        MAR_en = MIR[7],
        MDR_en = MIR[8],
         PC_en = MIR[9],
         SP_en = MIR[10],
         LV_en = MIR[11],
```

```
        CPP_en = MIR[12],
        TOS_en = MIR[13],
        OPC_en = MIR[14],
          H_en = MIR[15],
      alu_ctrl = MIR[21:16],
    shift_ctrl = MIR[23:22],
          JAMZ = MIR[24],
          JAMN = MIR[25],
          JMPC = MIR[26],
  next_address = MIR[pL+27:27];

// output enable registers, not real registers
reg MDR_oe,
    PC_oe,
    MBR_s_oe,
    MBR_u_oe,
    SP_oe,
    LV_oe,
    CPP_oe,
    TOS_oe,
    OPC_oe;

// load MIR @negedge clk
always @(negedge clk or posedge rst) begin
    if ( rst ) begin
            MIR <= 0;
    end else
            MIR <= wMIR;
end

// compute the high_bit of next address
always @(JAMN or JAMZ or next_address[pL] or N_flag or Z_flag)
    case ( {JAMN, JAMZ} )
    2'b00 : high_bit = next_address[pL];
    2'b10 : high_bit = next_address[pL] | N_flag;
    2'b01 : high_bit = next_address[pL] | Z_flag;
    2'b11 : high_bit = next_address[pL] | N_flag | Z_flag;
    endcase

// load MPC @posedge clk
always @(posedge clk or posedge rst) begin
    if (rst)
            MPC <= 0;
    else begin
            // load new MPC:
            if ( JMPC )
                    MPC <= { high_bit, (next_address[pL-1:0] | MBR) };
            else
                    MPC <= { high_bit, next_address[pL-1:0] };
    end
end

//
// output enable decoder
//
always @( MIR[3:0] ) begin
    // first set all output enable lines to zero:
    MDR_oe = 1'b0;
    PC_oe = 1'b0;
    MBR_s_oe = 1'b0;
    MBR_u_oe = 1'b0;
    SP_oe = 1'b0;
    LV_oe = 1'b0;
    CPP_oe = 1'b0;
    TOS_oe = 1'b0;
    OPC_oe = 1'b0;

    // decode the output enable control lines:
    case ( MIR[3:0] )
    4'b0000 : MDR_oe   = 1'b1;
    4'b0001 : PC_oe    = 1'b1;
    4'b0010 : MBR_s_oe = 1'b1;
    4'b0011 : MBR_u_oe = 1'b1;
    4'b0100 : SP_oe    = 1'b1;
    4'b0101 : LV_oe    = 1'b1;
    4'b0110 : CPP_oe   = 1'b1;
    4'b0111 : TOS_oe   = 1'b1;
```

```
    4'b1000 : OPC_oe   = 1'b1;
    default :
            ; // do nothing
    endcase
end

endmodule
```

mem_ctrl.v

```
module mem_ctrl
(
    clk,
    rst,

    addr_bus,
    data_bus,
    data_rd, data_wr,
    r_data_rd, r_data_wr,

    pc_bus,
    inst_bus,
    inst_rd, inst_wr,
    r_inst_rd, r_inst_wr,

    // generated clock
    clk_out,

    // SRAM interface
    ADDR, SRAM_CS, OE, WE, MDBUF_DIR, MDBUF_OE, BS, DATA,

    // IO interface
    IO_IN, IO_OUT
);

input clk;
input rst;

input [15:0] addr_bus;

inout [15:0] data_bus;
input data_rd, data_wr;
output r_data_rd, r_data_wr;

input [15:0] pc_bus;

reg [15:0] r_addr_bus;
reg r_data_rd, r_data_wr;

inout [7:0] inst_bus;
input inst_rd, inst_wr;
output r_inst_rd, r_inst_wr;

reg [15:0] r_pc_bus;
reg r_inst_rd, r_inst_wr;

output clk_out;

//////////////////////////////////////////////////////
// MEMORY INTERFACE
output [23:0] ADDR;
output SRAM_CS;
output OE;
output WE;
output MDBUF_DIR;
output MDBUF_OE;
output [3:0] BS;
inout [31:0] DATA;

// IO INTERFACE
input [7:0] IO_IN;
output [7:0] IO_OUT;

reg WE;

assign SRAM_CS=1'b0;
assign OE=1'b0;
assign MDBUF_OE=1'b0;
assign MDBUF_DIR=~WE;
assign BS=4'b0000;

reg [3:0] s;
reg [3:0] n_s;
```

```
reg clk_out;

reg [15:0] data_bus_out;
reg [7:0] inst_bus_out;

reg [15:0] r_data_bus;
reg [7:0] r_inst_bus;

// state constants
parameter s_s =4'b0000;
parameter s_p0=4'b0001;
parameter s_p1=4'b0010;
parameter s_p2=4'b0011;
parameter s_p3=4'b0100;
parameter s_d0=4'b1001;
parameter s_d1=4'b1010;
parameter s_d2=4'b1011;
parameter s_d3=4'b1100;
parameter s_d =4'b1111;

assign inst_bus = (r_inst_rd==1'b1) ? inst_bus_out : 8'bz;
assign data_bus = (r_data_rd==1'b1) ? data_bus_out : 16'bz;

assign ADDR = ( ( (s==s_p0) || (s==s_p1) || (s==s_p2) || (s==s_p3) ) && ( (r_inst_rd == 1'b1) ||
(r_inst_wr==1'b1) ) ) ? {5'b00000,1'b0,r_pc_bus,2'b00} : 24'bz;
assign ADDR = ( ( (s==s_d0) || (s==s_d1) || (s==s_d2) || (s==s_d3) ) && ( (r_data_rd == 1'b1) ||
(r_data_wr==1'b1) ) ) ? {5'b00000,1'b1,r_addr_bus,2'b00} : 24'bz;

assign DATA = ((s==s_p2) && (r_inst_wr==1'b1)) ? {r_inst_bus,r_inst_bus,r_inst_bus,r_inst_bus} :
32'bz;
assign DATA = ((s==s_d2) && (r_data_wr==1'b1)) ? {r_data_bus,r_data_bus} : 32'bz;

reg [7:0] IO_OUT;

////////////////////////////////////////////////////////

// main sequence
always @(posedge clk or posedge rst)
    if (rst)
            s <= s_d;
    else
            s <= n_s;

// register op parameters
always @(posedge clk or posedge rst)
    if (rst) begin
            {r_data_rd,r_data_wr,r_inst_rd,r_inst_wr} <= 4'b0;
            {r_pc_bus,r_addr_bus} <= {16'b0,16'b0};
            {r_data_bus,r_inst_bus} <= {16'b0,8'b0};
    end
    else begin
            if (s==s_s) begin
                    {r_data_rd,r_data_wr,r_inst_rd,r_inst_wr} <= {data_rd,data_wr,inst_rd,inst_wr};
                    {r_pc_bus,r_addr_bus} <= {pc_bus,addr_bus};

                    if (data_wr)
                            r_data_bus <= data_bus;

                    if (inst_wr)
                            r_inst_bus <= inst_bus;
            end
    end

// mem/io read process
always @(posedge clk or posedge rst)
    if (rst) begin
            data_bus_out <= 16'b0;
            inst_bus_out <= 8'b0;
    end
    else begin
            // read from program mem
            if ((s==s_p3) && (r_inst_rd==1'b1))
                    inst_bus_out <= DATA[7:0];

            // write data mem/io to data_bus_out (io@0xffff)
            if ((s==s_d3) && (r_data_rd==1'b1))
```

```
                    data_bus_out <= (r_addr_bus==16'hffff) ? {8'b0,IO_IN} : DATA[15:0];
    end

// io write process
always @(posedge clk or posedge rst)
    if (rst) begin
            IO_OUT <= 8'b0;
    end
    else begin
            if ((s==s_d3) && (r_addr_bus==16'hffff) && (r_data_wr==1'b1))
                    IO_OUT <= r_data_bus;
    end

// next state and clk_out
always @(s)
    case(s)
    s_s  : {n_s,clk_out}   = {s_p0,1'b1};
    s_p0 : {n_s,clk_out}   = {s_p1,1'b1};
    s_p1 : {n_s,clk_out}   = {s_p2,1'b1};
    s_p2 : {n_s,clk_out}   = {s_p3,1'b0};
    s_p3 : {n_s,clk_out}   = {s_d0,1'b0};
    s_d0 : {n_s,clk_out}   = {s_d1,1'b0};
    s_d1 : {n_s,clk_out}   = {s_d2,1'b0};
    s_d2 : {n_s,clk_out}   = {s_d3,1'b0};
    s_d3 : {n_s,clk_out}   = {s_d, 1'b0};
    s_d  : {n_s,clk_out}   = {s_s, 1'b0};

    default: {n_s,clk_out} = {s_s, 1'b0};
    endcase

// WE process
always @(s or r_data_rd or r_data_wr or r_inst_rd or r_inst_wr)
    case(s)
    s_s  : WE = 1'b1;
    s_p0 : WE = (r_inst_wr) ? 1'b1 : 1'b1;
    s_p1 : WE = (r_inst_wr) ? 1'b0 : 1'b1;
    s_p2 : WE = (r_inst_wr) ? 1'b0 : 1'b1;
    s_p3 : WE = (r_inst_wr) ? 1'b1 : 1'b1;
    s_d0 : WE = (r_data_wr) ? 1'b1 : 1'b1;
    s_d1 : WE = (r_data_wr) ? 1'b0 : 1'b1;
    s_d2 : WE = (r_data_wr) ? 1'b0 : 1'b1;
    s_d3 : WE = (r_data_wr) ? 1'b1 : 1'b1;
    s_d  : WE = 1'b1;
    default: WE = 1'b1;
    endcase

endmodule
```

ROM.v

```
module ROM(a,o);

input [8:0] a;
output [35:0] o;
reg [35:0] o;
always @(a) begin
case (a)
//              --------- --- -- ------ --------- --- ----
//              XXXXXXXXX JJJ SS FFEEII HOTCLSPMM WRF BBBB
//              NEXT_ADDR MAA LR 01NNNN  POPVPCVA REE 3210
//                        PMM LA   ABVC  CSP   RR IAC
//                        CNZ 81     A           TDT
//                                                 H
//              --------- --- -- ------ --------- --- ----
9'h000 : o = 36'b000000010_000_00_000000_000000000_000_0000; //  goto 0x2
9'h001 : o = 36'b001000000_000_00_110101_000000100_000_0001; //  PC=PC+1;goto 0x40
9'h002 : o = 36'b000000000_100_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto (MBR)
9'h003 : o = 36'b000000100_000_00_010100_100000000_000_0111; //  H=TOS;goto 0x4
9'h004 : o = 36'b000000010_000_00_111100_001000010_100_0000; //  TOS=MDR=H+MDR;wr;goto 0x2
9'h005 : o = 36'b000000110_000_00_010100_100000000_000_0111; //  H=TOS;goto 0x6
9'h006 : o = 36'b000000010_000_00_111111_001000010_100_0000; //  TOS=MDR=MDR-H;wr;goto 0x2
9'h007 : o = 36'b000001000_000_00_010100_100000000_000_0111; //  H=TOS;goto 0x8
9'h008 : o = 36'b000000010_000_00_001100_001000010_100_0000; //  TOS=MDR=H AND MDR;wr;goto 0x2
9'h009 : o = 36'b000001010_000_00_010100_100000000_000_0111; //  H=TOS;goto 0xA
9'h00a : o = 36'b000000010_000_00_011100_001000010_100_0000; //  TOS=MDR=H OR MDR;wr;goto 0x2
9'h00b : o = 36'b000000010_000_00_010100_000000010_100_0111; //  MDR=TOS;wr;goto 0x2
9'h00c : o = 36'b000001101_000_00_000000_000000000_000_0000; //  goto 0xD
9'h00d : o = 36'b000000010_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x2
9'h00e : o = 36'b000001111_000_00_010100_000000001_000_0100; //  MAR=SP;goto 0xF
9'h00f : o = 36'b000010001_000_00_010100_100000000_100_0000; //  H=MDR;wr;goto 0x11
9'h010 : o = 36'b000010110_000_00_110101_000001001_000_0100; //  SP=MAR=SP+1;goto 0x16
9'h011 : o = 36'b000010010_000_00_010100_000000010_000_0111; //  MDR=TOS;goto 0x12
9'h012 : o = 36'b000010100_000_00_110110_000000001_100_0100; //  MAR=SP-1;wr;goto 0x14
9'h013 : o = 36'b000100111_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x27
9'h014 : o = 36'b000000010_000_00_011000_001000000_000_0000; //  TOS=H;goto 0x2
9'h015 : o = 36'b000011000_000_00_010100_100000000_000_0101; //  H=LV;goto 0x18
9'h016 : o = 36'b000010111_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x17
9'h017 : o = 36'b000000010_000_00_010100_001000010_100_0010; //  TOS=MDR=MBR;wr;goto 0x2
9'h018 : o = 36'b000011001_000_00_111100_000000001_010_0011; //  MAR=H+MBRU;rd;goto 0x19
9'h019 : o = 36'b000011010_000_00_110101_000001001_000_0100; //  SP=MAR=SP+1;goto 0x1A
9'h01a : o = 36'b000011011_000_00_110101_000000100_101_0001; //  PC=PC+1;wr;fetch;goto 0x1B
9'h01b : o = 36'b000000010_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x2
9'h01c : o = 36'b000011101_000_00_111100_000000001_000_0011; //  MAR=H+MBRU;goto 0x1D
9'h01d : o = 36'b000011110_000_00_010100_000000010_100_0111; //  MDR=TOS;wr;goto 0x1E
9'h01e : o = 36'b000011111_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x1F
9'h01f : o = 36'b000100000_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x20
9'h020 : o = 36'b000000010_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x2
9'h021 : o = 36'b000100010_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x22
9'h022 : o = 36'b000100011_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x23
9'h023 : o = 36'b000011001_000_00_111100_000000001_010_0101; //  MAR=H+LV;rd;goto 0x19
9'h024 : o = 36'b000100101_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x25
9'h025 : o = 36'b000100110_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x26
9'h026 : o = 36'b000011101_000_00_111100_000000001_000_0101; //  MAR=H+LV;goto 0x1D
9'h027 : o = 36'b000101000_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x28
9'h028 : o = 36'b000101001_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x29
9'h029 : o = 36'b000011001_000_00_111100_000000001_010_0110; //  MAR=H+CPP;rd;goto 0x19
9'h02a : o = 36'b000101011_000_00_111100_000000001_010_0011; //  MAR=H+MBRU;rd;goto 0x2B
9'h02b : o = 36'b000101100_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x2C
9'h02c : o = 36'b000101101_000_00_010100_100000000_000_0000; //  H=MDR;goto 0x2D
9'h02d : o = 36'b000101110_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x2E
9'h02e : o = 36'b000000010_000_00_111100_000000010_100_0010; //  MDR=H+MBR;wr;goto 0x2
9'h02f : o = 36'b000110000_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x30
9'h030 : o = 36'b000110001_000_10_010100_100000000_000_0010; //  H=MBR<<8;goto 0x31
9'h031 : o = 36'b000110010_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x32
9'h032 : o = 36'b000110011_000_00_111100_000000100_001_1000; //  PC=H+OPC;fetch;goto 0x33
9'h033 : o = 36'b000000010_000_00_000000_000000000_000_0000; //  goto 0x2
9'h034 : o = 36'b000110101_000_00_010100_010000000_000_0111; //  OPC=TOS;goto 0x35
9'h035 : o = 36'b000110111_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x37
9'h036 : o = 36'b000011100_000_00_010100_100000000_000_0101; //  H=LV;goto 0x1C
9'h037 : o = 36'b000000001_000_00_010100_000000000_000_1000; //  N=OPC;if (N) goto 0x101; else goto 0x1
9'h038 : o = 36'b000111001_000_00_010100_010000000_000_0111; //  OPC=TOS;goto 0x39
9'h039 : o = 36'b000111010_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x3A
9'h03a : o = 36'b000000001_011_00_010100_000000000_000_1000; //  Z=OPC;if (Z) goto 0x101; else goto 0x1
```

```
9'h03b : o = 36'b000111100_000_00_110110_000001001_000_0100; //  SP=MAR=SP-1;goto 0x3C
9'h03c : o = 36'b000111101_000_00_010100_100000000_010_0000; //  H=MDR;rd;goto 0x3D
9'h03d : o = 36'b000111110_000_00_010100_010000000_000_0111; //  OPC=TOS;goto 0x3E
9'h03e : o = 36'b000111111_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x3F
9'h03f : o = 36'b000000001_011_00_111111_000000000_000_1000; //  Z=OPC-H;if (Z) goto 0x101; else goto 0x1
9'h040 : o = 36'b001000001_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x41
9'h041 : o = 36'b000000010_000_00_000000_000000000_000_0000; //  goto 0x2
9'h042 : o = 36'b001000011_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x43
9'h043 : o = 36'b001000100_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x44
9'h044 : o = 36'b001000101_000_00_111100_000000001_010_0110; //  MAR=H+CPP;rd;goto 0x45
9'h045 : o = 36'b001000110_000_00_110101_010000000_000_0001; //  OPC=PC+1;goto 0x46
9'h046 : o = 36'b001000111_000_00_010100_000000100_001_0000; //  PC=MDR;fetch;goto 0x47
9'h047 : o = 36'b001001000_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x48
9'h048 : o = 36'b001001001_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x49
9'h049 : o = 36'b001001010_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x4A
9'h04a : o = 36'b001001011_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x4B
9'h04b : o = 36'b001001100_000_00_111111_001000000_000_0100; //  TOS=SP-H;goto 0x4C
9'h04c : o = 36'b001001101_000_00_110101_001000001_000_0111; //  TOS=MAR=TOS+1;goto 0x4D
9'h04d : o = 36'b001001110_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x4E
9'h04e : o = 36'b001001111_000_10_010100_100000000_000_0011; //  H=MBRU<<8;goto 0x4F
9'h04f : o = 36'b001010000_000_00_011100_100000000_000_0011; //  H=H OR MBRU;goto 0x50
9'h050 : o = 36'b001010001_000_00_111101_000000010_100_0100; //  MDR=H+SP+1;wr;goto 0x51
9'h051 : o = 36'b001010010_000_00_010100_000001001_000_0000; //  SP=MAR=MDR;goto 0x52
9'h052 : o = 36'b001010011_000_00_010100_000000010_100_1000; //  MDR=OPC;wr;goto 0x53
9'h053 : o = 36'b001010100_000_00_110101_000001001_000_0100; //  SP=MAR=SP+1;goto 0x54
9'h054 : o = 36'b001010101_000_00_010100_000000010_100_0101; //  MDR=LV;wr;goto 0x55
9'h055 : o = 36'b001010110_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x56
9'h056 : o = 36'b000000010_000_00_010100_000010000_000_0111; //  LV=TOS;goto 0x2
9'h057 : o = 36'b000001100_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0xC
9'h058 : o = 36'b001011010_000_00_000000_000000000_000_0000; //  goto 0x5A
9'h059 : o = 36'b000001011_000_00_110101_000001001_000_0100; //  SP=MAR=SP+1;goto 0xB
9'h05a : o = 36'b001011011_000_00_010100_000010001_010_0000; //  LV=MAR=MDR;rd;goto 0x5B
9'h05b : o = 36'b001011100_000_00_110101_000000001_000_0101; //  MAR=LV+1;goto 0x5C
9'h05c : o = 36'b001011101_000_00_010100_000000100_011_0000; //  PC=MDR;rd;fetch;goto 0x5D
9'h05d : o = 36'b001011110_000_00_010100_000000001_000_0100; //  MAR=SP;goto 0x5E
9'h05e : o = 36'b001100001_000_00_010100_000010000_000_0000; //  LV=MDR;goto 0x61
9'h05f : o = 36'b000001110_000_00_110110_000000001_010_0100; //  MAR=SP-1;rd;goto 0xE
9'h060 : o = 36'b000000011_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x3
9'h061 : o = 36'b000000010_000_00_010100_000000010_100_0111; //  MDR=TOS;wr;goto 0x2
9'h062 : o = 36'b001100011_000_00_010100_000000010_100_0111; //  MDR=TOS;wr;goto 0x63
9'h063 : o = 36'b001100101_000_00_000000_000000000_000_0000; //  goto 0x65
9'h064 : o = 36'b000000101_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x5
9'h065 : o = 36'b001100110_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x66
9'h066 : o = 36'b001100111_000_00_000000_000000000_000_0000; //  goto 0x67
9'h067 : o = 36'b000000010_000_00_010100_001000000_000_0000; //  TOS=MDR;goto 0x2
9'h068 : o = 36'b001101001_000_00_110101_000001001_000_0100; //  SP=MAR=SP+1;goto 0x69
9'h069 : o = 36'b000000010_000_00_010100_001000000_100_0000; //  TOS=MDR;wr;goto 0x2
9'h06a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h06b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h06c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h06d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h06e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h06f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h070 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h071 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h072 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h073 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h074 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h075 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h076 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h077 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h078 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h079 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h07a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h07b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h07c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h07d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h07e : o = 36'b000000111_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x7
9'h07f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h080 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h081 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h082 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h083 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h084 : o = 36'b000101010_000_00_010100_100000000_000_0101; //  H=LV;goto 0x2A
9'h085 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h086 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h087 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
```

```
9'h088 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h089 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h08f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h090 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h091 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h092 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h093 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h094 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h095 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h096 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h097 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h098 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h099 : o = 36'b000111000_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x38
9'h09a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h09b : o = 36'b000110100_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x34
9'h09c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h09d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h09e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h09f : o = 36'b000111011_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x3B
9'h0a0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a7 : o = 36'b000101111_000_00_110110_010000000_000_0001; //  OPC=PC-1;goto 0x2F
9'h0a8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0a9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0aa : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ab : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ac : o = 36'b001011000_000_00_010100_000001001_010_0101; //  SP=MAR=LV;rd;goto 0x58
9'h0ad : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ae : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0af : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b0 : o = 36'b000001001_000_00_110110_000001001_010_0100; //  SP=MAR=SP-1;rd;goto 0x9
9'h0b1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b6 : o = 36'b001000010_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x42
9'h0b7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0b9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ba : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0bb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0bc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0bd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0be : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0bf : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c4 : o = 36'b100000000_100_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto (MBR OR 0x100)
9'h0c5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0c9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ca : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0cb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0cc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0cd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ce : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0cf : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
```

```
9'h0d5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0d9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0da : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0db : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0dc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0dd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0de : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0df : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0e9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ea : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0eb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ec : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ed : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ee : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ef : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0f9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0fa : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0fb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0fc : o = 36'b001101000_000_00_010010_000000001_010_0000; //  MAR=-1;rd;goto 0x68
9'h0fd : o = 36'b001100010_000_00_010010_000000001_000_0000; //  MAR=-1;goto 0x62
9'h0fe : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h0ff : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h100 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h101 : o = 36'b000101111_000_00_110110_010000000_001_0001; //  OPC=PC-1;fetch;goto 0x2F
9'h102 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h103 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h104 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h105 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h106 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h107 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h108 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h109 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h10f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h110 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h111 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h112 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h113 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h114 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h115 : o = 36'b000100001_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x21
9'h116 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h117 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h118 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h119 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h11f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h120 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h121 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
```

```
9'h122 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h123 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h124 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h125 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h126 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h127 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h128 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h129 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h12f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h130 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h131 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h132 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h133 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h134 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h135 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h136 : o = 36'b000100100_000_00_110101_000000100_001_0001; //  PC=PC+1;fetch;goto 0x24
9'h137 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h138 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h139 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h13f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h140 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h141 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h142 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h143 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h144 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h145 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h146 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h147 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h148 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h149 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h14f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h150 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h151 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h152 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h153 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h154 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h155 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h156 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h157 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h158 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h159 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h15f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h160 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h161 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h162 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h163 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h164 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h165 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h166 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h167 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h168 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h169 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h16a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h16b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h16c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h16d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h16e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
```

```
9'h16f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h170 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h171 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h172 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h173 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h174 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h175 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h176 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h177 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h178 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h179 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h17f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h180 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h181 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h182 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h183 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h184 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h185 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h186 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h187 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h188 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h189 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h18f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h190 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h191 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h192 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h193 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h194 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h195 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h196 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h197 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h198 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h199 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19a : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19b : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19c : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19d : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19e : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h19f : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1a9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1aa : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ab : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ac : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ad : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ae : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1af : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1b9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ba : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1bb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
```

```
9'h1bc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1bd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1be : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1bf : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1c9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ca : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1cb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1cc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1cd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ce : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1cf : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1d9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1da : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1db : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1dc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1dd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1de : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1df : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1e9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ea : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1eb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ec : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ed : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ee : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ef : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f0 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f1 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f2 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f3 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f4 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f5 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f6 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f7 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f8 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1f9 : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1fa : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1fb : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1fc : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1fd : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1fe : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
9'h1ff : o = 36'b011111111_000_00_000000_000000000_000_0000; //  goto 0xFF
endcase
end
endmodule
```

ijvm_mon.v

```
module ijvm_mon
(
    CLK,
    RST,
    SER_IN,
    GO, SWITCH, LED,
    ADDR, SRAM_CS, OE, WE, MDBUF_DIR, MDBUF_OE, BS, DATA
);

// CLOCK & RESET
input CLK;
input RST;

// RS232 SER_IN
input SER_IN;

// PUSH BUTTON
input GO;
// IO
input [7:0] SWITCH;
output [7:0] LED;

// MEMORY SIGNALS
output [23:0] ADDR;
output SRAM_CS;
output OE;
output WE;
output MDBUF_DIR;
output MDBUF_OE;
output [3:0] BS;
inout [31:0] DATA;

// UART INTERNAL SIGNALS
wire baud9600;
wire x16_clk_w;
wire x16_clk;
wire [7:0] ser_data_in;
wire data_ready;

uart_clkgen clkgen1 (
    .clk(CLK),
    .rst(RST),
    .baud9600(baud9600),
    .x16_clk(x16_clk_w)
);

BUFG buf_x16 ( .I(x16_clk_w), .O(x16_clk) );

uart_rx uart2 (
    .rst(RST),
    .x16_clk(x16_clk),
    .data(ser_data_in),
    .ser_in(SER_IN),
    .data_ready(data_ready)
);

wire sys_clk_w;
wire sys_clk;

wire r_data_rd;
wire r_data_wr;

wire r_inst_rd;
wire r_inst_wr;

wire data_rd;
wire data_wr;

wire inst_rd;
wire inst_wr;

wire [15:0] addr_bus;
wire [15:0] data_bus;
```

```
wire [15:0] pc_bus;
wire [7:0] inst_bus;

wire [7:0] IO_IN;
wire [7:0] IO_OUT;

assign IO_IN=8'b0;
assign IO_OUT=8'bz;

mem_ctrl mem_ctrl_1
(
    .clk(CLK),
    .rst(RST),
    .addr_bus(addr_bus), .data_bus(data_bus),
    .data_rd(data_rd), .data_wr(data_wr),
    .r_data_rd(r_data_rd), .r_data_wr(r_data_wr),
    .pc_bus(pc_bus), .inst_bus(inst_bus),
    .inst_rd(inst_rd), .inst_wr(inst_wr),
    .r_inst_rd(r_inst_rd), .r_inst_wr(r_inst_wr),
    .clk_out(sys_clk_w),
    .ADDR(ADDR), .SRAM_CS(SRAM_CS), .OE(OE), .WE(WE), .MDBUF_DIR(MDBUF_DIR), .MDBUF_OE(MDBUF_OE),
.BS(BS), .DATA(DATA),
    .IO_IN(IO_IN), .IO_OUT(IO_OUT)
);

BUFG buf_sys_clk ( .I(sys_clk_w), .O(sys_clk) );

monitor monitor_1
(
    .sys_clk(sys_clk),
    .rst(RST),
    .GO(GO), .SWITCH(SWITCH), .LED(LED),

    .ser_data_in(ser_data_in),
    .data_ready(data_ready),

    .addr_bus(addr_bus), .data_bus(data_bus),
    .data_rd(data_rd), .data_wr(data_wr),
    .r_data_rd(r_data_rd), .r_data_wr(r_data_wr),
    .pc_bus(pc_bus), .inst_bus(inst_bus),
    .inst_rd(inst_rd), .inst_wr(inst_wr),
    .r_inst_rd(r_inst_rd), .r_inst_wr(r_inst_wr)
);

endmodule
```

monitor.v

```
`define ASCII_0 8'h30
`define ASCII_1 8'h31
`define ASCII_2 8'h32
`define ASCII_3 8'h33
`define ASCII_4 8'h34
`define ASCII_5 8'h35
`define ASCII_6 8'h36
`define ASCII_7 8'h37
`define ASCII_8 8'h38
`define ASCII_9 8'h39

`define ASCII_a 8'h61
`define ASCII_b 8'h62
`define ASCII_c 8'h63
`define ASCII_d 8'h64
`define ASCII_e 8'h65
`define ASCII_f 8'h66

`define ASCII_A 8'h41
`define ASCII_B 8'h42
`define ASCII_C 8'h43
`define ASCII_D 8'h44
`define ASCII_E 8'h45
`define ASCII_F 8'h46

`define ASCII_w 8'h77
`define ASCII_r 8'h72
`define ASCII_W 8'h57
`define ASCII_R 8'h52

//
// monitor
//
module monitor
(
    sys_clk,
    rst,
    GO, SWITCH, LED,

    ser_data_in,
    data_ready,

    addr_bus, data_bus,
    data_rd, data_wr,
    r_data_rd, r_data_wr,
    pc_bus, inst_bus,
    inst_rd, inst_wr,
    r_inst_rd, r_inst_wr
);

input sys_clk;
input rst;

input GO;
input [7:0] SWITCH;
output [7:0] LED;

input [7:0] ser_data_in;
input data_ready;

output [15:0] addr_bus;
inout [15:0] data_bus;

output data_rd, data_wr;
input r_data_rd, r_data_wr;

output [15:0] pc_bus;
inout [7:0] inst_bus;

output inst_rd, inst_wr;
input r_inst_rd, r_inst_wr;

reg data_rd, data_wr;
```

```
reg inst_rd, inst_wr;

reg Qa, nQb;

always @(posedge sys_clk or posedge rst)
    if (rst)
            Qa <= 0;
    else
            Qa <= data_ready;

always @(posedge sys_clk or posedge rst)
    if (rst)
            nQb <= 0;
    else
            nQb <= ~Qa;

assign r_data_ready = Qa && nQb;

function [3:0] hex2bin;
    input [7:0] digit;
    case(digit)
    `ASCII_0 : hex2bin=4'h0;
    `ASCII_1 : hex2bin=4'h1;
    `ASCII_2 : hex2bin=4'h2;
    `ASCII_3 : hex2bin=4'h3;
    `ASCII_4 : hex2bin=4'h4;
    `ASCII_5 : hex2bin=4'h5;
    `ASCII_6 : hex2bin=4'h6;
    `ASCII_7 : hex2bin=4'h7;
    `ASCII_8 : hex2bin=4'h8;
    `ASCII_9 : hex2bin=4'h9;

    `ASCII_a : hex2bin=4'ha;
    `ASCII_b : hex2bin=4'hb;
    `ASCII_c : hex2bin=4'hc;
    `ASCII_d : hex2bin=4'hd;
    `ASCII_e : hex2bin=4'he;
    `ASCII_f : hex2bin=4'hf;

    `ASCII_A : hex2bin=4'ha;
    `ASCII_B : hex2bin=4'hb;
    `ASCII_C : hex2bin=4'hc;
    `ASCII_D : hex2bin=4'hd;
    `ASCII_E : hex2bin=4'he;
    `ASCII_F : hex2bin=4'hf;
    default: hex2bin = 4'h0;
    endcase
endfunction

reg [3:0] s;
reg [2:0] s_c;

reg [3:0] T [0:7];

reg addr_t;

reg [15:0] pc_bus_out;
reg [7:0] inst_bus_out;

reg [15:0] addr_bus_out;
reg [15:0] data_bus_out;

parameter [3:0] s_0  = 4'b0000;

parameter [3:0] s_r0 = 4'b0001;
parameter [3:0] s_r1 = 4'b0010;
parameter [3:0] s_r2 = 4'b0011;
parameter [3:0] s_r3 = 4'b0100;

parameter [3:0] s_w0 = 4'b1001;
parameter [3:0] s_w1 = 4'b1010;
parameter [3:0] s_w2 = 4'b1011;
parameter [3:0] s_w3 = 4'b1100;

parameter [3:0] s_t0 = 4'b1111;

reg [15:0] led_out;
```

```
assign addr_bus = addr_bus_out; // THIS SHOULD BE MODIFIED
assign data_bus = (r_data_rd) ? 16'bz : data_bus_out;

assign pc_bus = pc_bus_out; // THIS SHOULD BE MODIFIED
assign inst_bus = (r_inst_rd) ? 8'bz : inst_bus_out;

assign LED = (GO) ? (SWITCH | led_out[15:8]) : led_out[7:0];

always @(posedge sys_clk or posedge rst)
    if (rst) begin
            s <= s_0;
            s_c <= 3'b000;
            addr_t <= 1'b0;
            led_out <= 16'b0;
            {inst_rd,inst_wr,data_rd,data_wr} <= 4'b0000;
            {addr_bus_out,data_bus_out} <= {16'hffff,16'hffff};
            {pc_bus_out,inst_bus_out} <= {16'hffff,8'hff};
    end
    else    begin
            case (s)
            s_0 :
                    if (r_data_ready)
                            case(ser_data_in)
                            "r" :
                                    {addr_t, s_c, s} <= {1'b0, 3'b011, s_r0}; // program
                            "R" :
                                    {addr_t, s_c, s} <= {1'b1, 3'b011, s_r0}; // data
                            "w" :
                                    {addr_t, s_c, s} <= {1'b0, 3'b101, s_w0}; // program
                            "W" :
                                    {addr_t, s_c, s} <= {1'b1, 3'b111, s_w0}; // data
                            "t" :
                                    {addr_t, s_c, s} <= {1'b0, 3'b000, s_t0};
                            default:
                                    {addr_t, s_c, s} <= {1'b0, 3'b000, s_0};
                            endcase
                    else
                            {addr_t, s_c, s} <= {1'b0, 3'b000, s_0};

            s_r0 :
                    if (r_data_ready) begin
                            T[s_c] <= hex2bin(ser_data_in);
                            if (s_c==3'b000) begin
                                    {data_rd,data_wr,inst_rd,inst_wr} <= 4'b0000;
                                    s <= s_r1;
                            end
                            else
                                    s_c <= s_c - 1;
                    end

            s_r1 : begin
                    if (addr_t) begin
                            data_rd <= 1'b1;
                            addr_bus_out <= {T[3],T[2],T[1],T[0]};
                    end
                    else begin
                            inst_rd <= 1'b1;
                            pc_bus_out <= {T[3],T[2],T[1],T[0]};
                    end
                    s <= s_r2;
            end

            s_r2 : begin
                    if (addr_t) begin
                            data_rd <= 1'b0;
                            led_out <= data_bus;
                    end
                    else begin
                            inst_rd <= 1'b0;
                            led_out <= {8'b0,inst_bus};
                    end
                    s <= s_r3;
            end

            s_r3 : begin
                    s <= s_0;
```

```
		end

		s_w0 :
			if (r_data_ready) begin
				T[s_c] <= hex2bin(ser_data_in);
				if (s_c==3'b000) begin
					{data_rd,data_wr,inst_rd,inst_wr} <= 4'b0000;
					s <= s_w1;
				end
				else
					s_c <= s_c - 1;
			end

		s_w1 : begin
			if (addr_t) begin
				data_wr <= 1'b1;
				addr_bus_out <= {T[7],T[6],T[5],T[4]};
				data_bus_out <= {T[3],T[2],T[1],T[0]};
			end
			else begin
				inst_wr <= 1'b1;
				pc_bus_out <= {T[5],T[4],T[3],T[2]};
				inst_bus_out <= {T[1],T[0]};
			end
			s <= s_w2;
		end

		s_w2 : begin
			if (addr_t) begin
				data_wr <= 1'b0;
			end
			else begin
				inst_wr <= 1'b0;
			end
			s <= s_w3;
		end

		s_w3 : begin
			s <= s_0;
		end

		s_t0 :
			if (r_data_ready) begin
				led_out <= {ser_data_in,ser_data_in};
				s <= s_0;
			end
		endcase
	end

endmodule
```

uart_clkgen.v

```
module uart_clkgen
(
    clk,
    rst,
    baud9600,
    x16_clk
);

input clk;
input rst;
output baud9600;
output x16_clk;

reg [15:0] x16count;
reg baud9600;
reg [3:0] baudcount;
reg x16_clk;

// Generate a communication clock and a sampling clock - 9600 baud
always @(posedge clk or posedge rst)
    if (rst == 1'b1) begin
            x16count <= 16'b0000000000000000 ;
            baud9600 <= 1'b0 ;
            baudcount <= 4'b0000 ;
            x16_clk <= 1'b0 ;
    end
    else begin
            //9600 = 130 (40MHz oscillator), 155 (48MHz osc) or 162 (50MHz osc)
            //if (x16count == 162) begin // 9600 @50MHz
            //if (x16count == 81) begin //19200 @50MHz
            if (x16count == 12) begin // 115k @50MHz
                    x16_clk <= ~x16_clk ;
                    x16count <= 16'b0000000000000000 ;
                    if (baudcount == 15) begin
                            baud9600 <= ~baud9600 ;
                            baudcount <= 4'b0000 ;
                    end
                    else
                            baudcount <= baudcount + 1 ;
            end
             else
                    x16count <= x16count + 1 ;
    end

endmodule
```

uart_rx.v

```
module uart_rx
(
    rst,
    x16_clk,
    data,
    ser_in,
    data_ready
);
   input rst; // rst
   input x16_clk; // This clock is 16 time as fast as baud rate clock
   output [7:0] data; // The output on these signals is valid on the rising DATA_READY signal
   input ser_in; // Serial input line
   output data_ready;

   reg [7:0] data;
   reg data_ready;

   reg [9:0] buf_xhdl0;
   reg ser_in1; // These signals are used to double buffer a signal that is sources external to
the board.
   reg ser_in2; // These signals are used to double buffer a signal that is sources external to
the board.
   // The double buffering syncs the signal to the on board clk.
   reg [3:0] sample_counter; // There should be 16 x16_clk samples for each bit in the word.
   reg [3:0] bit_count; // This counts the bits in the word.   1 start, 8 data, and 1 stop.
   reg valid_data; // Valid data word flag.

   parameter [2:0] idle = 0;
   parameter [2:0] wait_one = 1;
   parameter [2:0] wait_1st_half = 2;
   parameter [2:0] get_data = 3;
   parameter [2:0] wait_2nd_half = 4;
   parameter [2:0] last_bit = 5;

   reg [2:0] uart_state; //  Where in the transmitted word are we?

   always @(posedge x16_clk or posedge rst)
   begin
      if (rst == 1'b1) begin
         ser_in1 <= 1'b1 ;
         ser_in2 <= 1'b1 ;
         data <= 8'b00000000 ;
         data_ready <= 1'b0 ;
      end
      else begin
         ser_in1 <= ser_in ;
         ser_in2 <= ser_in1 ;
         data <= buf_xhdl0[8:1] ;
         data_ready <= valid_data ;
      end
   end

   always @(posedge x16_clk or posedge rst)
   begin
      if (rst == 1'b1)
      begin
         uart_state <= idle ;
         bit_count <= 4'b0000 ;
         sample_counter <= 4'b0000 ;
         buf_xhdl0 <= 10'b1111111111 ;
         valid_data <= 1'b0 ;
         bit_count <= 4'b0000 ;
      end
      else
      begin
         case (uart_state)
            idle :
                     begin
                        // Waiting for Start Bit
                        if (ser_in2 == 1'b0)
                           // Start new bit and new word
                           uart_state <= wait_1st_half ;
                        else
```

```
                    uart_state <= idle ;

                 sample_counter <= 4'b0001 ;
                 buf_xhdl0 <= 10'b1111111111 ;
                 valid_data <= 1'b0 ;
                 bit_count <= 4'b0000 ;
              end
        wait_one :
              begin
                 // Start of new 16 sample bit
                 uart_state <= wait_1st_half ;
                 buf_xhdl0 <= buf_xhdl0 ;
                 valid_data <= 1'b0 ;
                 bit_count <= bit_count ;
                 sample_counter <= 4'b0001 ;
              end
        wait_1st_half :
              begin
                 // wait 8 samples of bit
                 if (sample_counter == 4'b0111)
                    uart_state <= get_data ;
                 else
                    uart_state <= wait_1st_half ;

                 sample_counter <= sample_counter + 1'b1 ;
                 buf_xhdl0 <= buf_xhdl0 ;
                 valid_data <= 1'b0 ;
                 bit_count <= bit_count ;
              end
        get_data :
              begin
                 // get the ninth sample and call it good.
                 uart_state <= wait_2nd_half ;
                 buf_xhdl0 <= {ser_in2, buf_xhdl0[9:1]} ;
                 sample_counter <= sample_counter + 1'b1 ;
                 valid_data <= 1'b0 ;
                 bit_count <= bit_count ;
              end
        wait_2nd_half :
              begin
                 // wait the remaining samples.
                 if (sample_counter == 4'b1110)
                    uart_state <= last_bit ;
                 else
                    uart_state <= wait_2nd_half ;

                 sample_counter <= sample_counter + 1'b1 ;
                 buf_xhdl0 <= buf_xhdl0 ;
                 valid_data <= 1'b0 ;
                 bit_count <= bit_count ;
              end
        last_bit :
            begin
               if (bit_count == 4'b1001) begin // 9
                  // Is it last bit in word?
                  uart_state <= idle ;
                  bit_count <= 4'b0000 ;

                  // Last sample in Bit
                  if ((buf_xhdl0[9]) == 1'b1 & (buf_xhdl0[0]) == 1'b0)
                     valid_data <= 1'b1 ;
                  else
                     valid_data <= 1'b0 ;
               end
               else begin
                  uart_state <= wait_one ;
                  bit_count <= bit_count + 1'b1 ;
                  valid_data <= 1'b0 ;
               end
            end
      endcase
   end
end
endmodule
```

IJVM.ucf

```
##Main Clock (SYS_CLK)
NET "CLK" LOC = "C8";

##Data
NET "DATA<0>" LOC = "P4";
NET "DATA<1>" LOC = "R4";
NET "DATA<2>" LOC = "T3";
NET "DATA<3>" LOC = "T4";
NET "DATA<4>" LOC = "N5";
NET "DATA<5>" LOC = "P5";
NET "DATA<6>" LOC = "R5";
NET "DATA<7>" LOC = "T5";
NET "DATA<8>" LOC = "N6";
NET "DATA<9>" LOC = "P6";
NET "DATA<10>" LOC = "R6";
NET "DATA<11>" LOC = "T6";
NET "DATA<12>" LOC = "M6";
NET "DATA<13>" LOC = "N7";
NET "DATA<14>" LOC = "P7";
NET "DATA<15>" LOC = "R7";
NET "DATA<16>" LOC = "T7";
NET "DATA<17>" LOC = "M7";
NET "DATA<18>" LOC = "N8";
NET "DATA<19>" LOC = "P8";
NET "DATA<20>" LOC = "R8";
NET "DATA<21>" LOC = "P9";
NET "DATA<22>" LOC = "N9";
NET "DATA<23>" LOC = "T10";
NET "DATA<24>" LOC = "R10";
NET "DATA<25>" LOC = "P10";
NET "DATA<26>" LOC = "R11";
NET "DATA<27>" LOC = "T11";
NET "DATA<28>" LOC = "N10";
NET "DATA<29>" LOC = "M10";
NET "DATA<30>" LOC = "P11";
NET "DATA<31>" LOC = "R12";

##Address
NET "ADDR<0>" LOC = "G16";
NET "ADDR<1>" LOC = "H16";
NET "ADDR<2>" LOC = "J13";
NET "ADDR<3>" LOC = "J16";
NET "ADDR<4>" LOC = "J15";
NET "ADDR<5>" LOC = "J14";
NET "ADDR<6>" LOC = "K13";
NET "ADDR<7>" LOC = "K12";
NET "ADDR<8>" LOC = "L12";
NET "ADDR<9>" LOC = "K16";
NET "ADDR<10>" LOC = "K15";
NET "ADDR<11>" LOC = "K14";
NET "ADDR<12>" LOC = "L13";
NET "ADDR<13>" LOC = "L16";
NET "ADDR<14>" LOC = "L15";
NET "ADDR<15>" LOC = "L14";
NET "ADDR<16>" LOC = "M13";
NET "ADDR<17>" LOC = "M16";
NET "ADDR<18>" LOC = "M15";
NET "ADDR<19>" LOC = "M14";
NET "ADDR<20>" LOC = "N14";
NET "ADDR<21>" LOC = "N16";
NET "ADDR<22>" LOC = "N15";
NET "ADDR<23>" LOC = "P16";

### SRAM STUFF
NET "SRAM_CS" LOC = "A3";
NET "OE" LOC = "A4";
NET "WE" LOC = "C4";
NET "MDBUF_DIR" LOC = "E6";
NET "MDBUF_OE" LOC = "E7";
NET "BS<0>" LOC = "C6";
NET "BS<1>" LOC = "B6";
NET "BS<2>" LOC = "A6";
NET "BS<3>" LOC = "B7";
```

```
##LEDs
NET "LED<0>" LOC = "D16";
NET "LED<1>" LOC = "E15";
NET "LED<2>" LOC = "F14";
NET "LED<3>" LOC = "G13";
NET "LED<4>" LOC = "E16";
NET "LED<5>" LOC = "F15";
NET "LED<6>" LOC = "G14";
NET "LED<7>" LOC = "H13";

##Switches ( 0-7 DIP, 8&9 BUTTON)
NET "SWITCH<0>" LOC = "J1";
NET "SWITCH<1>" LOC = "K1";
NET "SWITCH<2>" LOC = "L1";
NET "SWITCH<3>" LOC = "L2";
NET "SWITCH<4>" LOC = "L3";
NET "SWITCH<5>" LOC = "M2";
NET "SWITCH<6>" LOC = "L4";
NET "SWITCH<7>" LOC = "L5";

#RENAMING
NET "RST" LOC = "N2";
```

ijvm_mon.ucf

```
##Main Clock (SYS_CLK)
NET "CLK" LOC = "C8";

##Data
NET "DATA<0>" LOC = "P4";
NET "DATA<1>" LOC = "R4";
NET "DATA<2>" LOC = "T3";
NET "DATA<3>" LOC = "T4";
NET "DATA<4>" LOC = "N5";
NET "DATA<5>" LOC = "P5";
NET "DATA<6>" LOC = "R5";
NET "DATA<7>" LOC = "T5";
NET "DATA<8>" LOC = "N6";
NET "DATA<9>" LOC = "P6";
NET "DATA<10>" LOC = "R6";
NET "DATA<11>" LOC = "T6";
NET "DATA<12>" LOC = "M6";
NET "DATA<13>" LOC = "N7";
NET "DATA<14>" LOC = "P7";
NET "DATA<15>" LOC = "R7";
NET "DATA<16>" LOC = "T7";
NET "DATA<17>" LOC = "M7";
NET "DATA<18>" LOC = "N8";
NET "DATA<19>" LOC = "P8";
NET "DATA<20>" LOC = "R8";
NET "DATA<21>" LOC = "P9";
NET "DATA<22>" LOC = "N9";
NET "DATA<23>" LOC = "T10";
NET "DATA<24>" LOC = "R10";
NET "DATA<25>" LOC = "P10";
NET "DATA<26>" LOC = "R11";
NET "DATA<27>" LOC = "T11";
NET "DATA<28>" LOC = "N10";
NET "DATA<29>" LOC = "M10";
NET "DATA<30>" LOC = "P11";
NET "DATA<31>" LOC = "R12";

##Address
NET "ADDR<0>" LOC = "G16";
NET "ADDR<1>" LOC = "H16";
NET "ADDR<2>" LOC = "J13";
NET "ADDR<3>" LOC = "J16";
NET "ADDR<4>" LOC = "J15";
NET "ADDR<5>" LOC = "J14";
NET "ADDR<6>" LOC = "K13";
NET "ADDR<7>" LOC = "K12";
NET "ADDR<8>" LOC = "L12";
NET "ADDR<9>" LOC = "K16";
NET "ADDR<10>" LOC = "K15";
NET "ADDR<11>" LOC = "K14";
NET "ADDR<12>" LOC = "L13";
NET "ADDR<13>" LOC = "L16";
NET "ADDR<14>" LOC = "L15";
NET "ADDR<15>" LOC = "L14";
NET "ADDR<16>" LOC = "M13";
NET "ADDR<17>" LOC = "M16";
NET "ADDR<18>" LOC = "M15";
NET "ADDR<19>" LOC = "M14";
NET "ADDR<20>" LOC = "N14";
NET "ADDR<21>" LOC = "N16";
NET "ADDR<22>" LOC = "N15";
NET "ADDR<23>" LOC = "P16";

### SRAM STUFF
NET "SRAM_CS" LOC = "A3";
NET "OE" LOC = "A4";
NET "WE" LOC = "C4";
NET "MDBUF_DIR" LOC = "E6";
NET "MDBUF_OE" LOC = "E7";
NET "BS<0>" LOC = "C6";
NET "BS<1>" LOC = "B6";
NET "BS<2>" LOC = "A6";
NET "BS<3>" LOC = "B7";
```

```
##RS232 Signals
NET "SER_IN" LOC = "A9";  #RS232RX

##LEDs
NET "LED<0>" LOC = "D16";
NET "LED<1>" LOC = "E15";
NET "LED<2>" LOC = "F14";
NET "LED<3>" LOC = "G13";
NET "LED<4>" LOC = "E16";
NET "LED<5>" LOC = "F15";
NET "LED<6>" LOC = "G14";
NET "LED<7>" LOC = "H13";
##Switches ( 0-7 DIP, 8&9 BUTTON)
NET "SWITCH<0>" LOC = "J1";
NET "SWITCH<1>" LOC = "K1";
NET "SWITCH<2>" LOC = "L1";
NET "SWITCH<3>" LOC = "L2";
NET "SWITCH<4>" LOC = "L3";
NET "SWITCH<5>" LOC = "M2";
NET "SWITCH<6>" LOC = "L4";
NET "SWITCH<7>" LOC = "L5";
#NET "SWITCH<8>" LOC = "N2";
#NET "SWITCH<9>" LOC = "N3";

#RENAMING
NET "RST" LOC = "N2";
NET "GO" LOC = "N3";
```