RV16JS - Arquitetura de Processador Experimental RISC-V 16-bit

Resumo - Este documento define características e formato de instruções de um processador que preserva os principais recursos e convenções da arquitetura Risc-V, mas simplificado para fins didáticos, com apenas 16 bits e 64 KB de memória, de forma a ser facilmente observável e controlável em um simulador implementado em JavaScript, e adequado para ensino de diferentes aspectos de arquitetura, organização, programação *assembly* e conceitos de E/S e S.Os.

Características:

- largura de dados e endereços de 16-bits;
- memória de 64 KBytes endereçada a byte;
- leitura e escrita de bytes ou de palavras de 16-bits em formato *little-endian* (LSB primeiro);
- 10 registradores gerais x0 a x9 e mais 6 registradores especiais, ou de controle;
- acesso unificado a todos os 16 registradores pelas mesmas instruções;

Registradores:

x0 : constante 0 - quando usado como destino, resultado é descartado;
x1 ou ra : registrador onde endereço de retorno de sub-rotina é guardado (convenção);
x2..x8 : registradores de uso genérico;
x9 ou sp : último registrador geral, usado como ponteiro para topo da pilha (convenção);
PC (x10) : *Program Counter* - indica próxima instrução a ser executada;
ST (x11) : *STatus register* - bits de controle da CPU (modo, ints, etc...);
IR (x12) : *Instruction Register* - armazena opcode (16 bits) da instrução sendo executada;
IM (x13) : *IMmediate* - guarda temporariamente imediato/*offset* para próxima instrução;
IV (x14) : *Interrupt Vector* - guarda endereço do tratador de interrupções (*ISR*, *kernel*);
IA (x15) : *Interrupt return Address* - valor de PC salvo e a ser restaurado em interrupções;

Formato Base - Todas as instruções ocupam 16-bits, iniciam sempre em endereço par, e usam o seguinte formato-base, dividido em 4 campos com tamanhos de 4, 4, 4, 4 bits:

[15-12] rd [11-8] rs1 [7-4] rs2 / offset / imm [3-0] *opcode*

Instruções - As 16 instruções fundamentais disponíveis pelo *opcode* de 4 bits são as seguintes:

ADD, ADDI, SHIFT, CONTROL,
SUB, AND, OR, XOR,
SW, LW, SB, LB,
BEQ, BLT, JAL, LUI,

Imediatos - Valores imediatos de 16 bits serão formados por uma instrução adicional precedente do tipo LUI - *load upper immediate*, que carrega os 12 bits mais significativos no registrador especial IM. O prefixo LUI é a única forma de acessar as versões com operando imediato (*I-type*) das instruções aritméticas e lógicas SUB, AND, OR, XOR, que se tornarão SUBI, ANDI, ORI and XORI. Após cada instrução que segue LUI, IM recebe 0 novamente, de forma que esse registrador é apenas temporário para compor instruções do tipo I de 32 bits de tamanho.

Variantes - algumas instruções possuem variantes, efetivamente utilizando um *opcode* de 5 bits, (5 bits menos significativos). O quinto bit pode ser usado quando a instrução não usa rs2, ou nos saltos, pois o bit menos significativo do *offset* deve ser 0, já que as instruções sempre iniciam em endereço par. Assim, as variantes de BEQ, BLT e JAL são BNE, BGE a JALR. Para as instruções de deslocamento e controle, mais variantes são possíveis, mas inicialmente serão implementadas com os *opcodes* de 5 bits apenas SRS/SLS (*Shift Right* e *Shift Left 1 Step*) e ECALL/IRET.

Extensões - extensões que sejam necessárias para ensino de tópicos e disciplinas específicas, como, por exemplo, outras instruções aritméticas, em ponto flutuante, vetoriais, etc..., podem ser implementadas com o opcode CONTROL, com os 6 últimos bits iguais a '1x0011'.

Tabela de Instruções - A seguinte tabela lista todas instruções e seu comportamento:

code	mnem	hex	type	operandos	execução	comentário
_0000	ADD	0	R	rd,rs1,rs2	x[rd] ← x[rs1] + x[rs2]	2's complement
_0001	ADDI	1	I	rd,rs1,imm	x[rd] ← x[rs1] + (IM | imm)	may have LUI before
00010	SRS	02	S	rd, rs1	x[rd] ← x[rs1] >> 1	shift right 1-bit step
10010	SLS	12	S	rd, rs1	x[rd] ← x[rs1] << 1	shift left 1-bit step
00011	ECALL	03	C	-	IA ← PC; PC ← IV;	Sw Int or SysCall
10011	IRET	13	C	-	PC ← IA;	Interrupt Return
_0100	SUB	4	R	rd,rs1,rs2	x[rd] ← x[rs1] - x[rs2]	2's complement
_0101	AND	5	R	rd,rs1,rs2	x[rd] ← x[rs1] & x[rs2]	bitwise and
_0110	OR	6	R	rd,rs1,rs2	x[rd] ← x[rs1] | x[rs2]	bitwise or
_0111	XOR	7	R	rd,rs1,rs2	x[rd] ← x[rs1] ^ x[rs2]	bitwise xor
_0100	SUBI	4	I	rd,rs1,imm	x[rd] ← x[rs1] - (IM | imm)	coded as LUI + SUB
_0101	ANDI	5	I	rd,rs1,imm	x[rd] ← x[rs1] & (IM | imm)	coded as LUI + AND
_0110	ORI	6	I	rd,rs1,imm	x[rd] ← x[rs1] | (IM | imm)	coded as LUI + OR
_0111	XORI	7	I	rd,rs1,imm	x[rd] ← x[rs1] ^ (IM | imm)	coded as LUI + XOR
_1000	SW	8	I	rd,rs1,offset	MemWord(x[rs1] + offset) ← x[rd]	Store 16-bit Word
_1001	LW	9	I	rd,rs1,offset	x[rd] ← MemWord(x[rs1] + offset)	Load 16-bit Word
_1010	SB	A	I	rd,rs1,offset	MemByte(x[rs1] + offset) ← x[rd]	Store Byte
_1011	LB	B	I	rd,rs1,offset	x[rd] ← MemByte(x[rs1] + offset)	Load Byte
01100	BEQ	0C	J	rd,rs1,offset	if (x[rd] == x[rs1]) PC += (IM | offset&E)	Branch if EQual
11100	BNE	1C	J	rd,rs1,offset	if (x[rd] != x[rs1]) PC += (IM | offset&E)	Branch if Not Equal
01101	BLT	0D	J	rd,rs1,offset	if (x[rd] < x[rs1]) PC += (IM | offset&E)	Branch if Less Than
11101	BGE	1D	J	rd,rs1,offset	if (x[rd] >= x[rs1]) PC += (IM | offset&E)	Br if Greater or Equal
01110	JAL	0E	J	rd,offset	x[rd] ← PC; PC += (IM | offset&E)	Jump And Link
11110	JALR	1E	J	rd,rs1,offset	x[rd] ← PC; PC ← x[rs1]+(IM | offset&E)	Jump And Link Reg
_1111	LUI	F	U	imm	IM ← imm	imm = IR & 0xFFF0

Tipos de Opcode: - apesar de todos as instruções usarem um formato semelhante, dividido em 4 campos de 4 bits, há algumas pequenas variações. A seguinte tabela faz uma definição mais precisa de todos esses casos, e usa a terminologia derivada historicamente de MIPS e RiscV.

tipo	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
tipo R - registradores	rd				rs1				rs2				opcode			
tipo I - imediato e memória	rd				rs1				imm/offset[3..0]				opcode			
tipo S - deslocamentos	rd				rs1				x	x	fun2		opcode			
tipo C - controle	não usado, ou reservado para extensões										fun2		opcode			
tipo J - saltos JAL e JALR	rd				rs1 (exceto JAL)				offset[3..1]			f1	opcode			
tipo U - prefixo LUI	12 bits mais significativos para próxima instrução												opcode			

NOP e HLT - As instruções NOP e HLT (*halt*) são pseudo-instruções, ou mnemônicos alternativos. NOP é sinônimo para ADD x0,x0,x0, que não tem nenhum efeito na máquina, e coincide com o *opcode* 0x0000 característico de memória vazia. HLT deve ser codificada com o *opcode* 0x00EE, de JAL x0,-2, que corresponde a um salto incondicional de volta para a mesma instrução, formando um laço infinito. Neste caso, o processador deve reconhecer essa condição e indicar o encerramento da execução;

Instruções com imediatos - as instruções SUBI..XORI estão listadas na tabela pois são instruções reconhecidas pelo processador, com um comportamento diferente de SUB..XOR, não usando o registrador rs2, e não apenas uma extensão do tamanho do valor imediato.

Pseudo-Instruções - O montador pode reconhecer alguns outros mnemônicos para facilitar a programação, traduzindo-os para as instruções presentes no processador. Além de NOP e HLT, e dos mnemônicos SUBI..XORI para as sequências LUI+SUB a LUI-XOR, poderão ser usados J para saltos, INC, DEC, entre outros, a serem definidos posteriormente.

Status Register ST - O registrador de estado da CPU deve controlar os modos, privilégios, e situação das interrupções, pelo menos, e portanto deve incluir os seguintes bits:

R - real or protected mode
K - kernel or user mode in case protected is on
I - interrupts enabled disabled
P - hardware interrupt pending
X - exception (illegal instruction/access)
S - environment/system call (= SW interrupt)

* numero da HW_INT pode ser consultado em mem, representando
que o computador tem um controlador de ints conectado
* numero da SW_INT pode estar em registrador ou em bits do ST
* numero da exceção precisa de bits no ST

***Nota:**
Esta especificação é experimental, está incompleta e não foi totalmente verificada nesse momento.

Decisões de Projeto (FAQ) - algumas justificativas para decisões sobre alternativas de projeto:

1) *Por que 10 registradores gerais?* Porque 8 é um número um pouco limitado considerando que 3 já têm um uso especial ou implícito (constante 0, endereço de retorno para chamada de sub-rotina, e ponteiro para pilha) e a arquitetura *load-store* exige operandos em registrador. Por outro lado, 16 registradores gerais seria mais difícil de acompanhar na interface e no *log* de execução. Os 10 registradores podem ser representados por apenas um dígito decimal, reduzindo ainda mais o espaço necessário na interface e *log*;
2) Por que os registradores especiais são acessíveis a todas instruções? Isso permite ao processador em modo kernel salvar/restaurar todo estado da CPU sem instruções adicionais, o que garante que os opcodes de 4 bits (com variantes) sejam suficientes;
3) Por que o formato 4, 4, 4, 4? Além de acomodar os 16 registradores, o formato de 4 campos de 4 bits pode ser facilmente visualizado em 4 caracteres em hexadecimal, o que não é possível em outros formatos usados como campos de 3 bits para 8 registradores. Assim, você pode praticamente programar em binário sem não tiver um montador;
4) Por que instrução LUI usa registrador IM? A instrução LUI é necessária para formar imediatos ou deslocamentos de 16 bits, pré-carregando os 12 bits mais significativos desses valores. Nesta versão de 16-bits, LUI não poderia especificar um dos outros registradores para isso;
5) Porque existe um *opcode* para ADDI diferente dos demais? O uso de imediato nas outras operações SUB.. XOR exige a instrução "prefixo" LUI. Entretanto, isso significa que para incremento/decremento, o programador/compilador teria apenas duas opções: ou codificar LUI+ADD ou então manter o valor 1/-1 em um registrador para usar ADD de 16bits, e essas duas alternativas representam uma sobrecarga considerável para operações tão comuns em laços quanto incrementar/decrementar contadores e ponteiros. A inclusão de uma única instrução ADDI com deslocamento de 4 bits (-8..+7) permite fazer incrementos e decrementos de *bytes*, palavras de 16 ou 32 bits, mais eficientemente;
6) É possível usar um imediato sem LUI nas instruções SUB..XOR? Neste formato não, as instruções addi..xori não podem ser codificadas em 16-bits, e são o resultado da combinação com o prefixo LUI, efetivamente formando uma instrução de 32 bits. De fato, ela não pode ser interrompida, e o processador não vai executar uma *trap* (atender interrupção) entre uma LUI e a instrução codificada a seguir;
7) É possível fazer um salto (*branch*) sem LUI? Sim, é possível, embora extremamente limitado. Saltos sem um prefixo LUI estarão limitados a deslocamentos de -8 a +7 em relação ao PC. O processador vai estender o sinal dos 4 bits do campo de imediato para isso. Entretanto, a solução mais simples para o montador é codificar sempre a sequencia LUI + salto para atingir qualquer posição de destino, até por que se não fizer isso significa que algumas instruções de salto teriam 2 *bytes* e outras 4, o que dificulta consideravelmente calcular os endereços;
8) Por que ECALL e IRET? As instruções ECALL e IRET são necessárias para implementar proteção no sistema, passando o processador do modo usuário para o modo supervisor quando operando em modo protegido. Se não fosse por isso, a chamada de sistema poderia ser feita como chamada de sub-rotina (com JAL) e o retorno de qualquer interrupção, inclusive de *hardware*, poderia ser feita por JALR considerando que o registrador IA guarda o PC salvo em uma interrupção e está acessível para qualquer instrução;
9) Por que o bit de interrupção pendente (requisitada) estará em registrador e não em memória? Esse estado deve ser testado a cada instrução. Seria inflexível e caro usar o barramento (ou sistema de endereçamento) para fazer isso. O número da interrupção de *hardware* que ocorreu pode ser consultado em um endereço específico de memória, representando que a máquina é construída com um controlador de interrupções externo conectado como dispositivo de E/S, e o *kernel* ou ISR é escrito em *software*, mas a linha de interrupção deve

ser um pino separado no processador e é testada pela FSM do processador, que fica independente de endereços fixos;

10) Por que IV e IA são registradores e não posições de memória? Consultar o IV e salvar o endereço de retorno em IA fazem parte da execução do processador, e ter registradores para isso deixa o sistema de E/S livre para ser definido em qualquer endereço;