Introducción

El procesador **ERV25** es un procesador de alta performance con estructura pipeline de 5 etapas.

Resumen de algunas de las características básicas del **ERV25**:

- Arquitectura de datos interna de 32 bits
- 32 registros de 32 bits. 31 registros de uso general (`x1-x31`), `x0` con valor `0` constante
- Memoria de programa separada de la memoria de datos
- Espacio de memoria de 32 bits con acceso indexado de a 8 bits
- Palabras de instrucción de 32 bits
- Set de instrucciones RISC-V RV32I simplificado
    - Soporte de TRAP de hardware trivial o desactivado
    - Soporte de instrucciones de nivel de maquina únicamente (sin privilegios, machine level)
    - Sin instrucciones SYSTEM
    - Sin CSR (registros de sistema y estado)
    - Sin instrucciones FENCE de memoria (FENCE = NOP)

RISC V es una arquitectura de set de instrucciones (ISA) de estándar abierto, basada en los principios de procesadores RISC. Si bien RISC V define únicamente el set de instrucciones, establece ciertas características del procesador que son necesarias para que el conjunto de instrucciones sea viable, como se especifica anteriormente.

Para la implementación de este procesador se recomienda acceder a las especificaciones de la ISA RISC-V: https://riscv.org/specifications/ratified/. En particular, leer de antemano los primeros dos capítulos de ***Volume 1, Unprivileged Specification***: 1. INTRODUCCIÓN y 2. SET DE INSTRUCCIONES RV32I (que se encuentra en las especificaciones RATIFIED).

Opcionalmente, para familiarizarse más con la arquitectura RISC-V, se puede referenciar ***Volume 2, Privileged Specification***, específicamente los capítulos 2 (CONTROL AND STATUS REGISTERS) y 3 (MACHINE-LEVEL ISA), aunque estos no son necesarios para la especificación básica del procesador **ERV25**.

Parte I - Especificaciones

Set de Instrucciones

La particularidad del set de instrucciones RISC-V es que el diseño del código binario de las instrucciones está pensado para facilitar la decodificación de las mismas por el hardware. De esta manera, se espera que la micro-instrucción, o señales de control, del procesador sean similares a los campos ya existentes en el código binario.

La ISA RISC-V RV32I utiliza seis formatos de instrucciones en base a los modos de direccionamiento de las mismas. El diseño de la ISA prioriza mantener los bits de ciertos campos (Registros de entrada, registro de salida, valor inmediato, etc.) en los mismos lugares en los distintos modos, para simplificar la decodificación.

Los dos bits menos significativos de codifican el tamaño de la palabra de instrucción. En el caso de la implementación base del **ERV25**, estos son siempre 11, ya que se implementa únicamente la extensión RV32I:

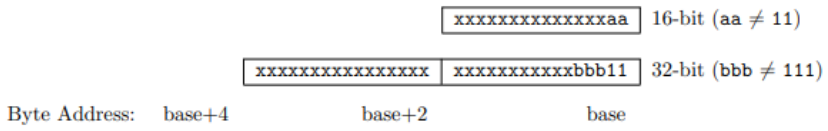

Figura 1.1 – codificación de largo de instrucción

Debido a la limitación del tamaño de valores inmediatos que se pueden almacenar en la palabra de instrucción (ya que la misma tiene 32 bits, igual que una palabra de datos), se deben usar dos instrucciones para cargar una constante de tamaño completo (32 bits). La mayor parte de las operaciones están limitadas a valores inmediatos de 11 bits.

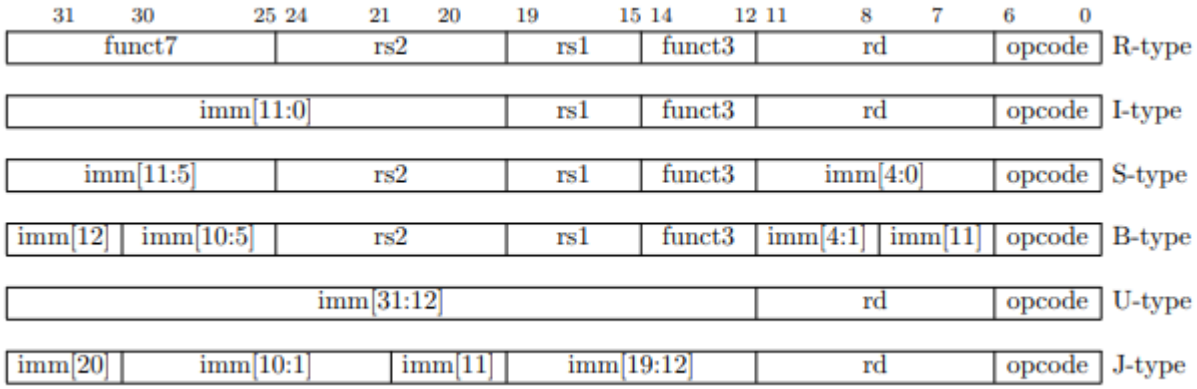

Figura 1.2 – formatos de instrucciones para RV32I

La mayor parte de las operaciones asumen valores signados, y para las constantes el procesador realiza extensión de signo (repite el valor del bit más significativo para completar los bits restantes).

Una descripción más en detalle, junto con algunas notas para la implementación, pueden encontrarse en las sección 2.2 (formatos de instrucción) y 2.3 (codificación de valores inmediatos) de la especificación de la ISA base RISC-V, paginas 11-13.

La **Tabla A** muestra los valores posibles para OPCODE. Recordar que `inst[1:0]` es siempre 11 dado que son instrucciones de 32 bits.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Tabla 1A – Resumen de OPCODEs posibles en RV32I

Para simplificar nuestra implementación, la instrucción FENCE, que se utiliza para sincronizar acceso de memoria en sistemas multihilo/multiprocesador, se puede implementar como un NOP. Asimismo, se permite que las instrucciones SYSTEM (ECALL, EBREAK y CSR*) no estén implementadas. Sin embargo, se recomienda considerar esta como una de las mejoras propuestas a la arquitectura mediante el soporte para TRAPS de hardware, junto con la implementación de los CSR de timers básicos requeridos por la especificación de RV32I (ver parte III)

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred | succ		00000	000	00000	0001111	FENCE
0000 | 0000 | 0000		00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Tabla 1B – set de instrucciones RV32I

Los tipos de instrucciones se resumen a continuación. Para una descripción detallada, ver la sección correspondiente del manual de la ISA RISC-V.

Computacionales (*ver Seccion 2.4*)

- ADD/SUB (suma y resta)
- Shift Left (SLL logico, SLA aritmetico)
- Shift Right (SRL logico, SRA aritmetico)
- Booleanas (AND, OR, XOR)
- Comparacion SLT (y SLTU no signada): rd = 1 si rs1 < rs2, 0 sino.

Transferencia de control (saltos – *ver sección 2.5*)

- No condicionales
  - JAL – salto relativo usando el valor inmediato como offset del PC, múltiplo de 2 bytes
  - JALR – salto indirecto tal que PC = `[rs1 + imm]`, con el bit menos significativo = 0
- Condicionales, equivalen a JAL pero solo saltan si se da la condición
  - BEQ/BNE – `rs1 == rs2` o `rs1 != rs2`
  - BLT/BLTU – `rs1 < rs2`
  - BGE/BGEU – `rs1 >= rs2`

Acceso a memoria (*ver sección 2.6*)

- LOAD
  - LW – carga en rd 32 bits desde la posición de memoria [`rs1 + imm`]
  - LH – carga en rd 16 bits desde la posición de memoria [`rs1 + imm`] (bit de signo extendido)
  - LHU – carga en rd 16 bits desde la posición de memoria [`rs1 + imm`] (sin extender bit signo)
  - LB/LBU – ídem LH pero de 8 bits
- STORE
  - SW – guarda los 32 bits de rs2 en [`rs1 + imm`]
  - SH, SHU, SB, SBU – análogos a Lx[U], guarda 8 o 16 bits signados/no signados

Banco de Registros

Se dispone de un banco de 31 registros para uso general más el registro `x0`. El registro `x0` siempre devuelve `0`, y cualquier intento de almacenar en el mismo es ignorado.

Si bien los registros son de uso general, se recomienda al programador usar ciertos registros para funcionalidad especifica:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Para mejorar la funcionalidad como microcontrolador, se recomienda diseñar el procesador de tal manera que los registros x28-x31 a puertos de entrada/salida (unidireccionales o bidireccionales).

Acceso a memoria (LOAD/STORE)

Se debe implementar la funcionalidad de acceso a memoria de datos externa mediante las instrucciones LOAD (LW/LH/LB[U]) y STORE (SW/SH/SB[U]).

El acceso a memoria debe contemplar el uso de memoria con latencias de acceso mayores a un ciclo de clock. Para simplificar la implementación, asuma que dispone de una señal de `memory_ready` que indica que los datos de lectura son válidos (para operaciones de Read), o que la escritura termino y se puede realizar otra operación de memoria (para operaciones de Write).

Parte II – Implementación

Cada equipo de trabajo deberá proponer a la cátedra un diseño de arquitectura basado en las especificaciones del ERV25, con sus propias características y funcionalidades adicionales (ver parte III a continuación). Una vez aprobada por la cátedra esta especificación, deberán implementarlo en FPGA y demostrar su funcionamiento mediante uno o más programas que puedan ser modificados o parametrizados “en vivo” durante la demostración en el laboratorio. Se intenta limitar el empleo de módulos desarrollados íntegramente en Verilog o equivalentes que podrán usarse solamente en casos puntuales y previamente acordados con la cátedra. El enfoque general apunta a la implementación del procesador con bloques de funciones lógicas (registros, ALU, etc.).

Implementación “MVP-0”

Para comenzar la implementación, se recomienda primero buscar implementar la siguiente funcionalidad básica. La misma consiste únicamente de los siguientes módulos para conformar un *“proto-procesador”*, que si bien no tiene instrucciones de control de flujo (saltos), puede realizar operaciones aritméticas secuenciales.

- Memoria de programa
  - Utilizar memoria de 8 bits, pero con salida de 4 palabras/32 bits.
  - Se accede a las palabras de 32 bits “saltando de a 4” en el *address*

- Banco de registros
  - 31 registros R/W, registro `x0` = `0`
  - Dos puertos de lectura (input: `address rs1`, `address rs2`; output: `data rs1`, `data rs2`)
  - Un puerto de escritura (input: `address rd`, `data rd`, `write enable`)

- Program Counter
  - Registro de 32 bits

- Módulo *Fetch*
  - Incrementa el PC en 4 por cada ciclo de clock
  - Busca en la memoria el contenido de [PC]

- Módulo *Decode*
  - Genera los valores de `rs1`, `rs2`, `imm`, opcode, `funct3`, `funct7` según la instrucción recibida
    - Considerar los distintos formatos según la tabla a continuación
  - Genera también señales de control especificando que operación se va a realizar, y que operandos utilizar.
    - Guardar en registro (si/no)
    - Utiliza rs1 (si/no)
    - Utiliza rs2 (si/no)
    - Utiliza imm (si/no)
    - Salto (si/no)
    - Leer de memoria (si/no)
    - Escribir a memoria (si/no)
    - Etc.
  - Si bien para el “MVP-0” so lo se necesita el flag de guardar en registro, los restantes ayudan a la implementación completa.

- ALU
  - Recibe opcode, `funct3`, `funct7`, valor de `rs1`, valor de `rs2`, `imm`, y otros flags de decode si es necesario
  - Produce el resultado correcto según la especificación de la operación

- Data bus
    - Conecta los registros con la ALU

Se recomienda implementar este “MVP-0” sin pipeline, con todos los módulos combinacionales, de manera que los mismos se ejecuten en cascada dentro de un solo ciclo de clock.

Existen dos opciones para implementar el banco de registros:

1. En la implementación básica, el banco de registros actualiza los puertos de salida cada vez que cambian los valores en los address de entrada de rs1 y rs2. De esta manera el procesador completo funciona de manera combinacional, hasta el momento del “write back”, que ocurre en el siguiente flanco de clock
2. El banco de registros puede actualizar la salida de los valores de rs1 y rs2 con el flanco *descendente* del clock. Esto efectivamente crea un ciclo intermedio en el procesador. De usar esta opción, considerar la propagación de las otras señales.s

opcode	funct3	Instr Format
LUI	xxx	U
AUIPC	xxx	U
JAL	xxx	J
JALR	xxx	I
LOAD	xxx	I
OP-IMM	not[001,101]	I
OP-IMM	001 or 101	R
OP	xxx	R
STORE	xxx	S
BRANCH	xxx	B
MISC-MEM	xxx	I [N/A]
SYSTEM	xxx	I [N/A]

Tabla 2A – Formato de instrucciones
(Nota: solo se incluyen los opcodes del set RV32I reducido)

Se busca ver la funcionalidad del procesador cargando instrucciones aritméticas básicas de la memoria y mostrando el uso del banco de registros y de la ALU. El checkpoint 1 consiste en mostrar esta versión funcionando en simulaciones, y opcionalmente en hardware de FPGA.

Arquitectura pipeline prototípica

La figura 2.1 presenta *una posible* arquitectura esquemática del procesador **EVR24**, incluyendo el pipeline de 5 etapas. Se recuerda que se puede reemplazar el pipeline por otras mejoras, previo acuerdo con la cátedra.

Este diagrama está pensado para proveer los módulos básicos de la implementación, sin embargo, debe ser ajustado para implementar el procesador de acuerdo a las especificaciones.

Para la implementación de los cálculos de offset tanto para funciones de salto como para acceso a memoria, que suelen involucrar el resultado de `[imm + rs2]`, se puede o bien utilizar los sumadores de la ALU, o bien implementar un bloque independiente para esto. Ambas opciones tienen ventajas y desventajas, justifique las decisiones de diseño.

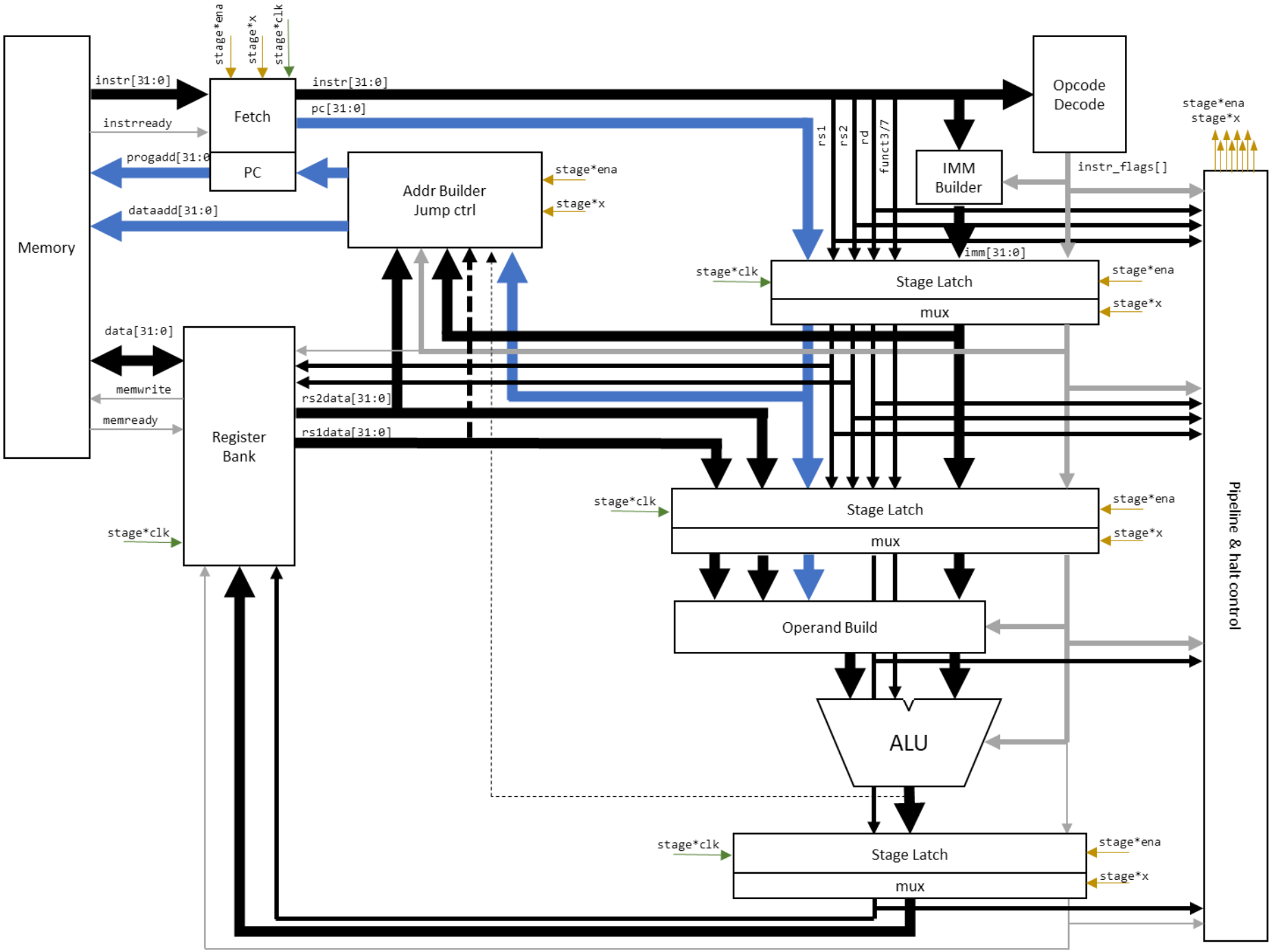

Figura 2.1 – Prototipo de arquitectura con pipeline de 5 etapas

Señales de clock

Para simplificar el proceso de implementación y prueba, se recomienda dejar clocks independientes en las distintas etapas. En la primera implementación, se puede prescindir del bloque de Halt y utilizar el procesador con señales de clock independientes, cada una fuera de fase 72^o (1/5) con respecto a la anterior. Con este esquema de clock las 5 etapas se ejecutan de manera secuencial dentro de un ciclo de clock.

Una vez comprobado el funcionamiento secuencial, se puede alinear todas las señales de clock (0^o de fase) para lograr que las distintas etapas ocurran al mismo tiempo.

Control de saltos

Este módulo en particular requiere de un diseño detallado, que debe considerar, entre otros factores:

- Resolución de la dirección relativa del salto
- Saltos condicionales
- Comportamiento del pipeline ante un salto
- Almacenamiento del PC original en el banco de registros (completar la operación)

Control de pipeline y halt

Al igual que el control de saltos, este módulo es crítico para la operación correcta del procesador. Debe considerar el comportamiento del pipeline ante ciertas condiciones, al menos:

- Comportamiento ante un salto (incondicional o condicional)
- Dependencias de registros (Read after Write)
- Acceso a memoria
- Cualquier otra condición de detención del pipeline (e.j. operaciones largas de la ALU)

Detallar que señales son necesarias hacia otros módulos para un correcto funcionamiento del procesador en todos los casos.

Se debe evitar que este módulo detenga el pipeline excesivamente (en todas o la mayor parte de las instrucciones), ya que de esta manera se pierde cualquier ventaja del diseño con pipeline.

Parte III - Mejoras a la implementación básica

La parte final de este trabajo consiste en agregar a su implementación una serie de mejoras a la arquitectura básica. A continuación, se encuentran algunas sugerencias, sin embargo, no son prescriptivas y cada equipo puede sugerir nuevas mejoras a la cátedra.

Puerto Serie

Esta extensión básica permite el uso de una consola serie (usualmente RS232) para poder realizar operaciones de I/O básicas con el procesador, por ejemplo con una consola sencilla.

Timers y Excepciones/Traps

El agegado de los CSRs minimos (3 timers) junto con el soporte de excepciones y traps permite utilizar compiladores comerciales para RISC-V, y – en teoría – permitiría correr software complejo que no necesite niveles de privilegios del compilador.

Para ellos, se debe implementar el hardware para el manejo de los saltos por TRAP, algunas de las excepciones básicas descriptas en las instrucciones de RV32I (por ejemplo, error de alineamiento al hacer un salto), y tres timers de 64 bits en el hardware:

- CYCLE: cuenta de ciclos
- TIME: cuenta de tiempo (ticks), se debe poder leer la constante que especifique segundos/tick
- INSTRET: número de instrucciones retiradas

Las siguientes instrucciones permiten acceso de lectura a estos timers:

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

Stack de hardware de punteros de retorno

Debido al formato de las instrucciones, el guardado del Program Counter a un registro puede ser codificado por el programador en la instrucción binaria sin necesidad de hardware adicional.

Sin embargo, para mejorar la performance se implementa un Stack de direcciones de retorno, basado en el uso de los registros x1 y x5.

Al detectar una llamada a función [JAL o JALR con rd=x1 o x5], se debe guardar el nuevo valor de rd en el stack.

Al detectar un retorno de función [JALR con rs1=x1 o x5], se debe retirar el valor de top of stack, utilizándolo para la dirección de salto. Nota: Se debe luego verificar que efectivamente el salto fue hacia la dirección de la predicción.

Se puede implementar el comportamiento completo de este predictor según la tabla 2.1 de la sección 2.5 del manual de la ISA RISC-V.

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	push and pop
link	*link*	1	push

Table 2.1: Return-address stack prediction hints encoded in register specifiers used in the instruction. In the above, *link* is true when the register is either `x1` or `x5`.

Extensiones del set de instrucciones

Se pueden implementar extensiones del set de instrucciones RV32I. Si bien estas pueden ser ya definidas por RISC-V, se recomienda buscar aplicaciones específicas (e.j. criptografía, manejo de matrices, DSP, etc.).

Salida VGA

Consiste en un módulo de salida de gráficos en baja o media resolución (320 x 200, 8 colores como mínimo). Se puede implementar como un periférico, CSR, o utilizando extensiones del set de instrucciones de RISC-V