Exploiting ILP

Yamin Li

Department of Computer Science
Faculty of Computer and Information Sciences
Hosei University, Tokyo 184-8584 Japan

http://cis.k.hosei.ac.jp/~yamin/

Parallelism Among Iterations of a Loop

- The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop.
- This type of parallelism is often called *loop-level parallelism*. Example:

```
for (i=0; i<1000; i++)
    x[i] = x[i] + s;
```

- Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap.

Basic Pipeline Scheduling

- We first translate the above loop example to MIPS assembly language.
- The MIPS code of the loop example looks like:

```
Loop:  L.D     F0, 0(R1)      ; F0: array element
       ADD.D   F4, F0, F2     ; adds scalar in F2
       S.D     0(R1), F4      ; stores result
       DADDI   R1, R1, #8     ; 8 bytes per DW
       DADDI   R2, R2, #−1    ; counter −1
       BNEZ    R2, Loop       ; branches R2!=zero
```

Execution Without Any Scheduling

			Clock cycle issued
Loop:	L.D	F0, 0(R1)	1
	stall		2
	ADD.D	F4, F0, F2	3
	stall		4
	stall		5
	S.D	0(R1), F4	6
	DADDI	R1, R1, #8	7
	DADDI	R2, R2, #−1	8
	BNEZ	R2, Loop	9
	stall		10

This requires 10 clock cycles per iteration.

Scheduling by Reorganizing Instructions

```
Loop:   L.D      F0, 0(R1)
        DADDI    R2, R2, #−1
        ADD.D    F4, F0, F2
        DADDI    R1, R1, #8
        BNEZ     R2, Loop        ; delayed branch
        S.D      −8(R1), F4      ; delay slot, executed always
```

Execution time has been reduced from 10 cycles to 6.

Loop Unrolling

There are four copies of loop body:

```
Loop:   L.D     F0, 0(R1)       ; 1
        DADD    F4, F0, F2
        S.D     0(R1), F4
        L.D     F6, 8(R1)       ; 2
        DADD    F8, F6, F2
        S.D     8(R1), F8
        L.D     F10, 16(R1)     ; 3
        DADD    F12, F10, F2
        S.D     16(R1), F12
        L.D     F14, 24(R1)     ; 4
        DADD    F16, F14, F2
        S.D     24(R1), F16
        DADDI   R2, R2, #-4     ; counter - 4
        DADDI   R1, R1, #32
        BNEZ    R2, Loop
```

Loop Unrolling and Scheduling

Execution time is reduced to 15/4, or 3.75 clock cycles.

```
Loop:   L.D      F0, 0(R1)      ; 1              1
        L.D      F6, 8(R1)      ; 2              2
        L.D      F10, 16(R1)    ; 3              3
        L.D      F14, 24(R1)    ; 4              4
        DADDI    R2, R2, #-4    ; counter - 4    5
        DADD     F4, F0, F2                      6
        DADD     F8, F6, F2                      7
        DADD     F12, F10, F2                    8
        DADD     F16, F14, F2                    9
        S.D      0(R1), F4                      10
        S.D      8(R1), F8                      11
        S.D      16(R1), F12                    12
        S.D      24(R1), F16                    13
        BNEZ     R2, Loop                       14
        DADDI    R1, R1, #32    ; delay slot    15
```

Software Pipeline

- We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code.
- *Software pipeline* is another important technique that have been developed for this purpose.
- *Software pipeline* is a technique for reorganizing loops such that each instruction in software-pipeline code is made from instructions chosen from *different iterations* of the original loop.
- Software pipeline doesn't unroll the loop, but it can be sought of as *symbolic* loop unrolling.

Software Pipeline — Example

- A software pipelined loop interleaves instructions from different iterations without unrolling the loop.
- Our example, shown below, contains one load, one add, and one store, each from a different iterations.

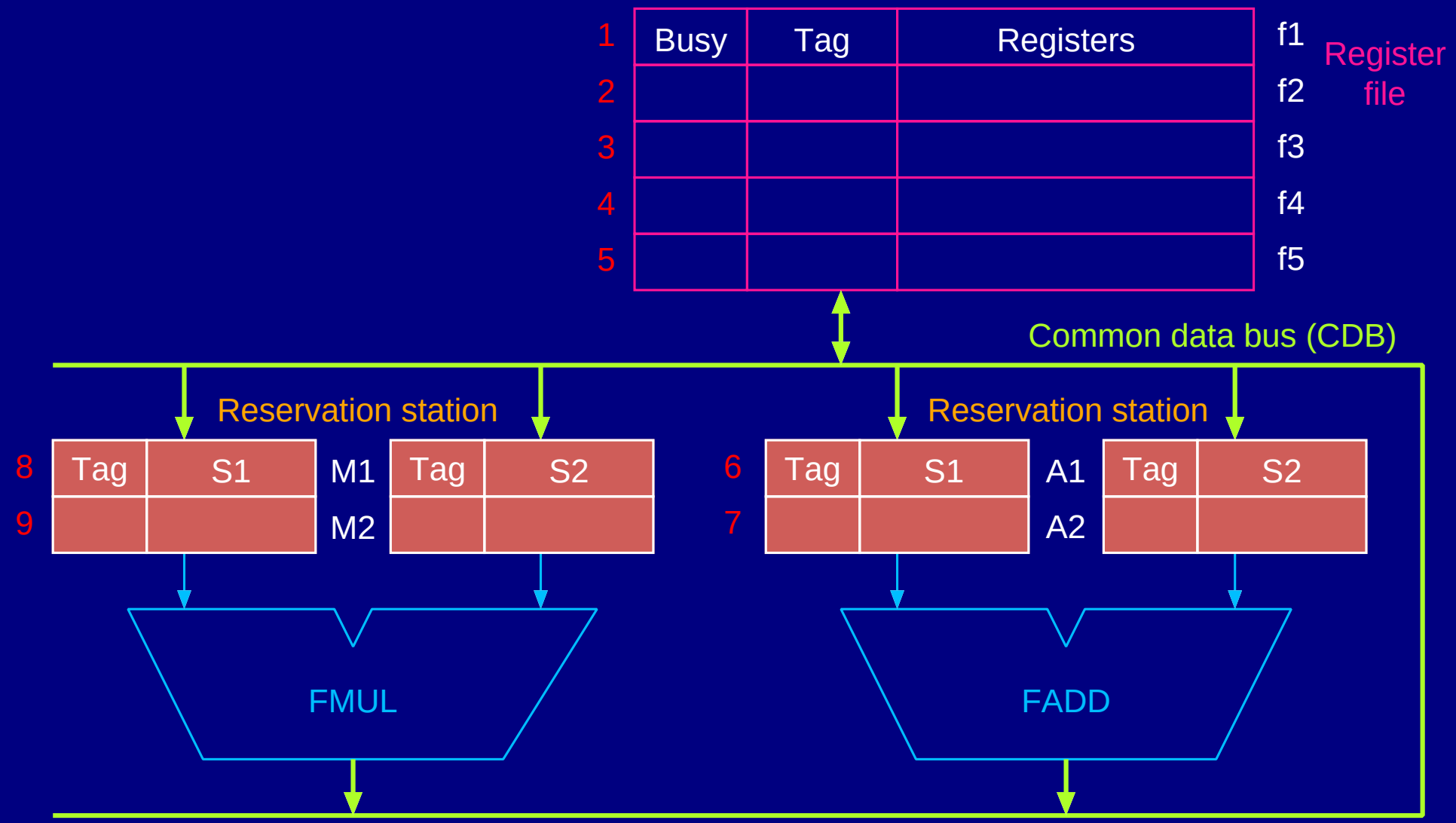
```
Loop:  S.D     −8(R1), F4     ; stores into X[i −1]
       ADD.D   F4, F0, F2     ; adds to X[i]
       L.D     F0, 8(R1)      ; loads X[i+1]
       DADDI   R2, R2, #−1
       BNEZ    R2, Loop
       DADDI   R1, R1, #8     ; delay slot
```

- Some start-up code and finish-up code are required.

Dynamic Instruction Scheduling

- Instruction scheduling
  - Static scheduling – Done by software at compile-time
  - Dynamic scheduling – Done by hardware at run-time
- Main feature of dynamic scheduling
  - Out-of-order execution
- Main methods of dynamic scheduling
  - Scoreboard
  - Tomasulo

Tomasulo Organization

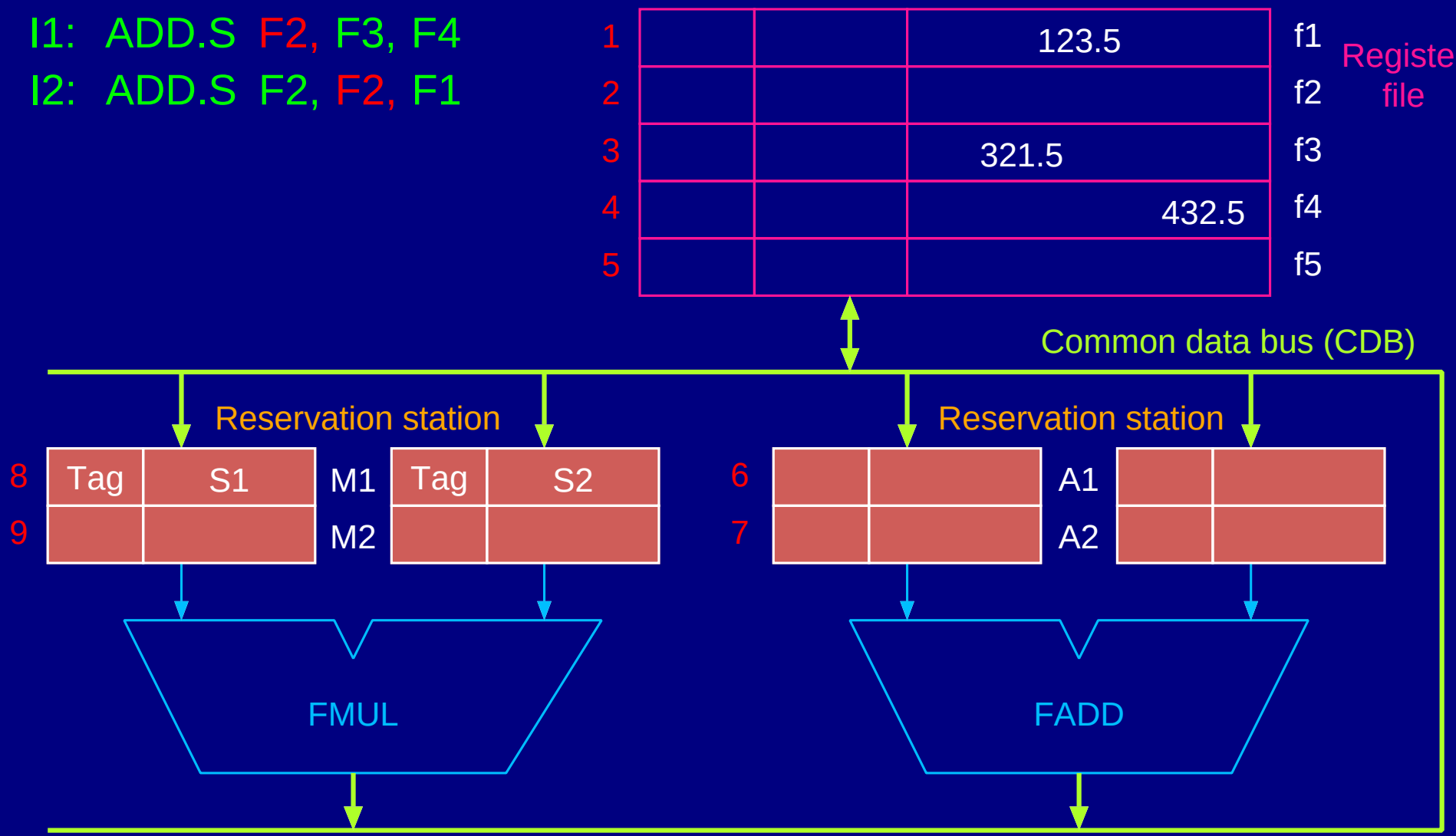


Three Stages of Tomasulo Algorithm

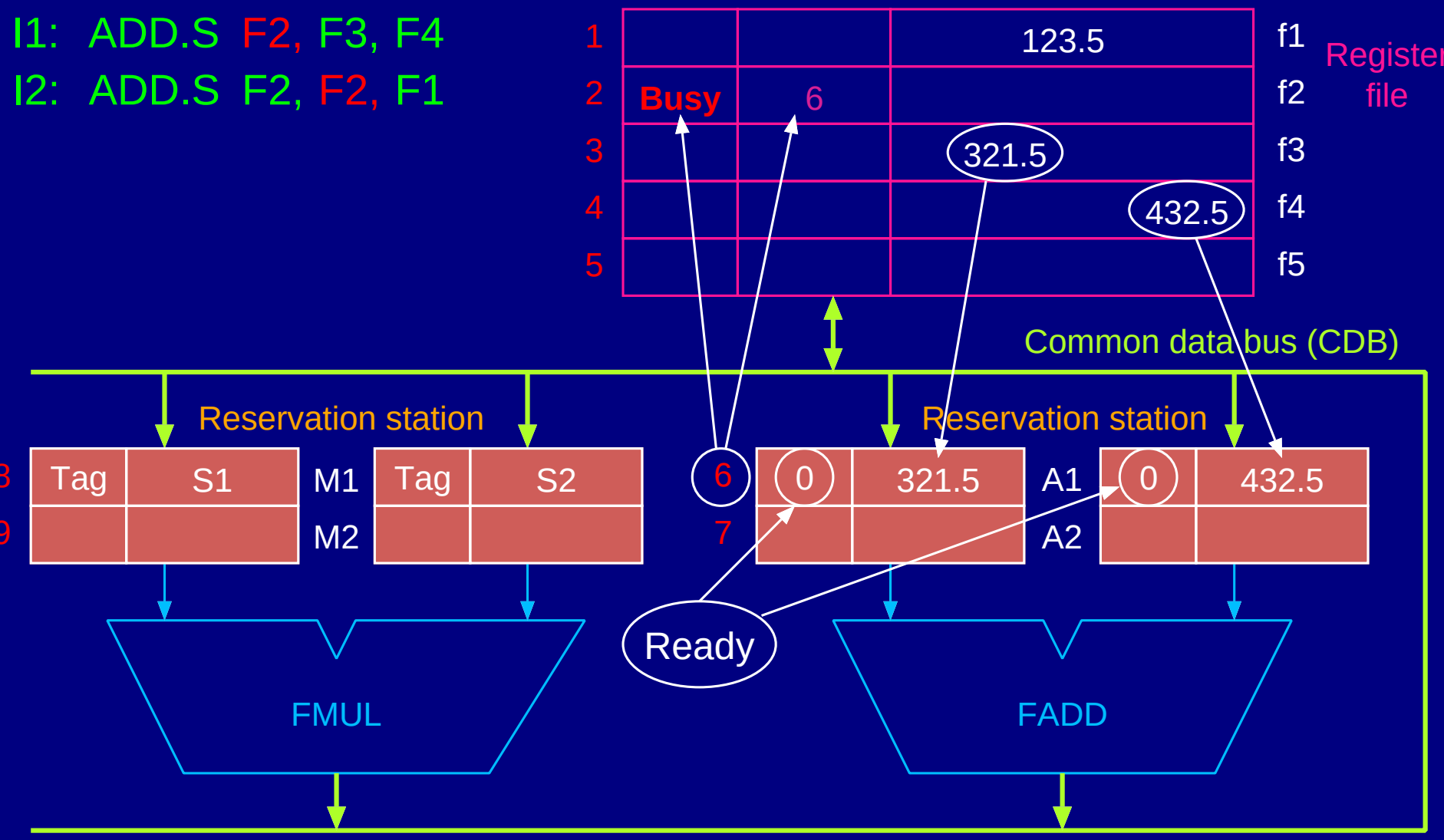
1. Issue — get instruction from FP operation queue
   - If reservationstation free (no structure hazard), control issues instruction and sends operands (renames registers).
2. Execution — operate on operans
   - When both operans ready then executes.
   - If not ready, watch Common Data Bus for result.
3. Write result — finish execution
   - Write on Common Data Bus to all awaiting units.
   - Mark reservation station available.

Tomasulo Algorithm Example

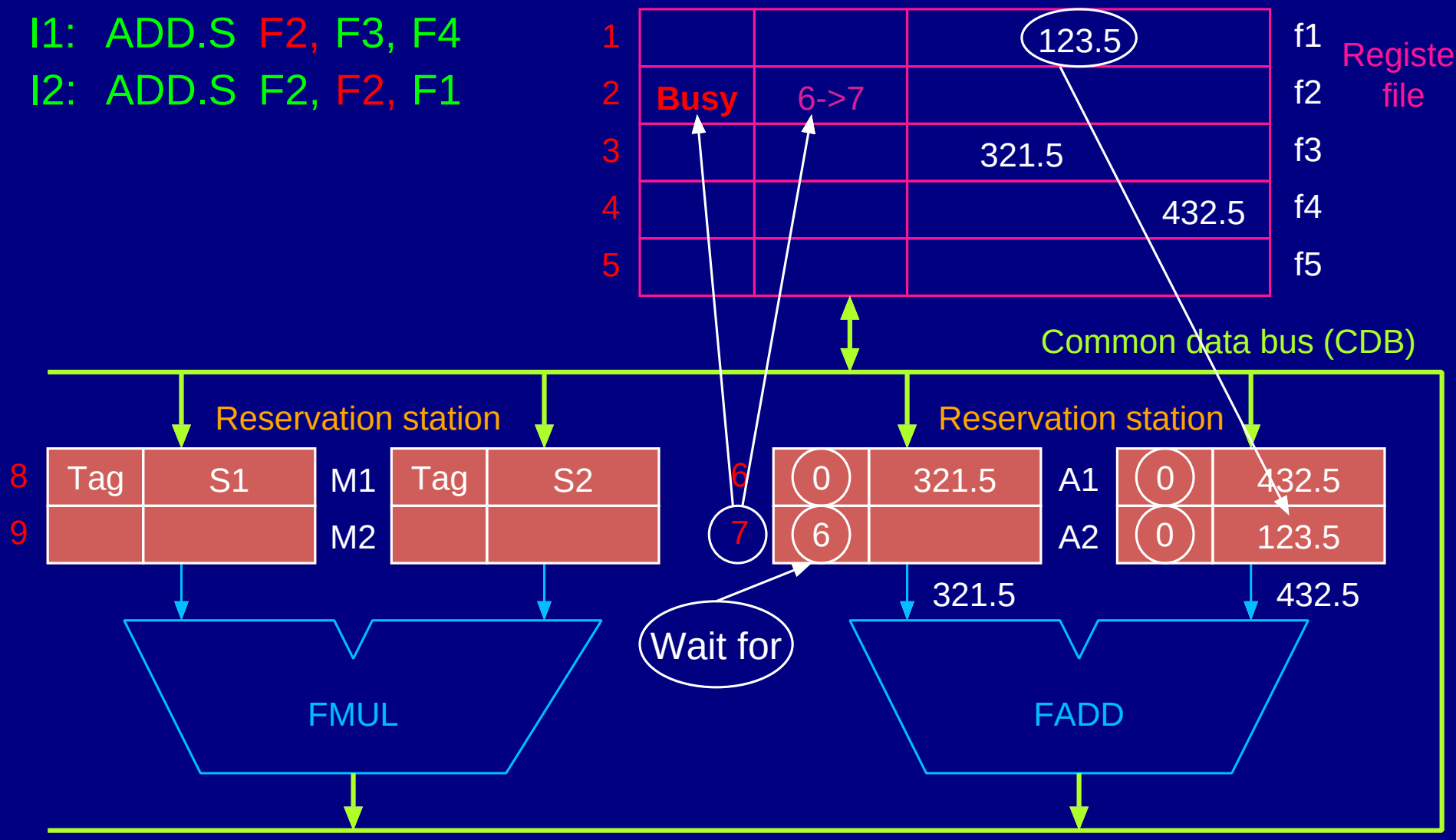
I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



Clock Cycle 1 — Issuing I1

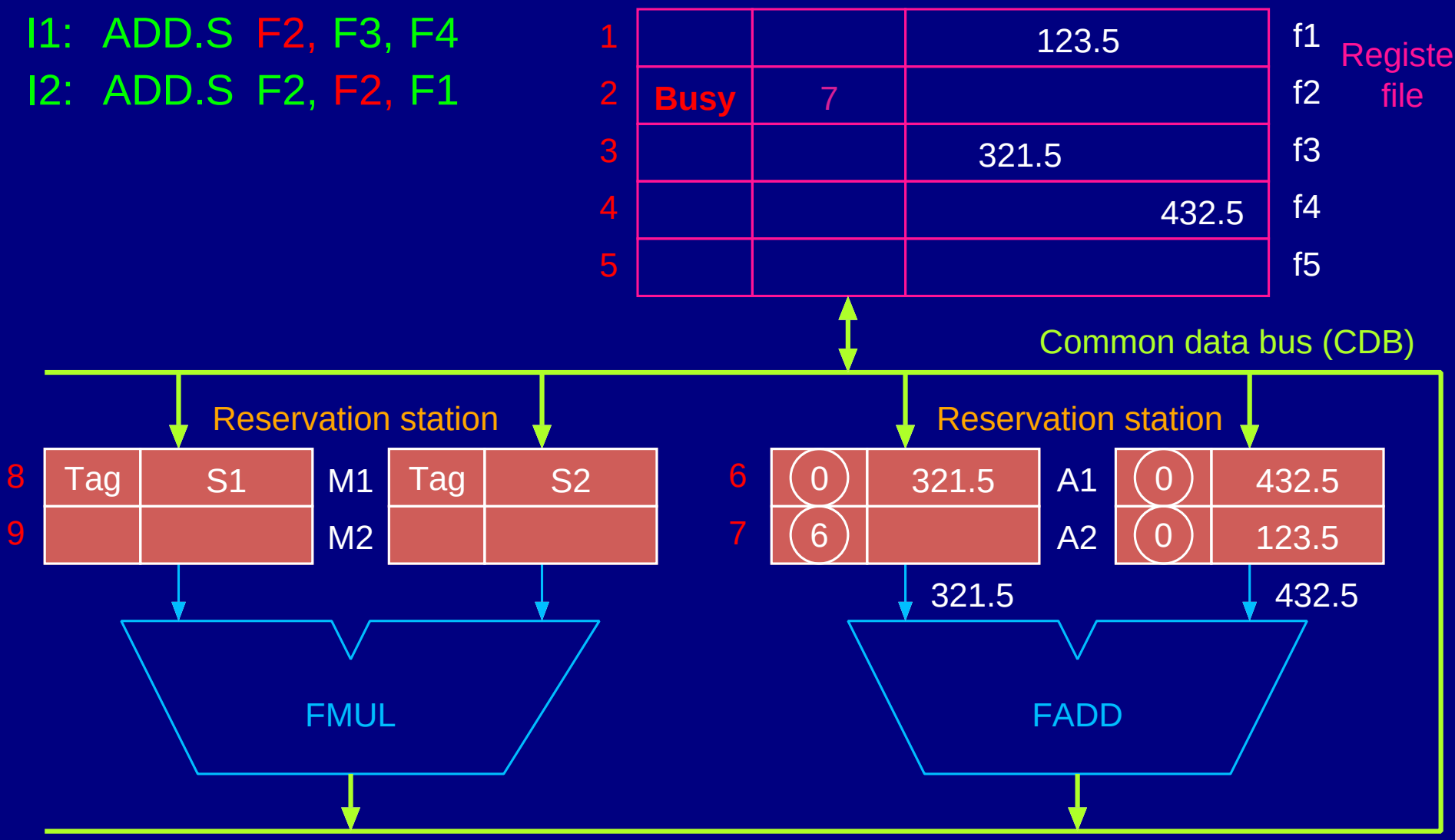


Clock Cycle 2 — Executing I1 (1), Issuing I2



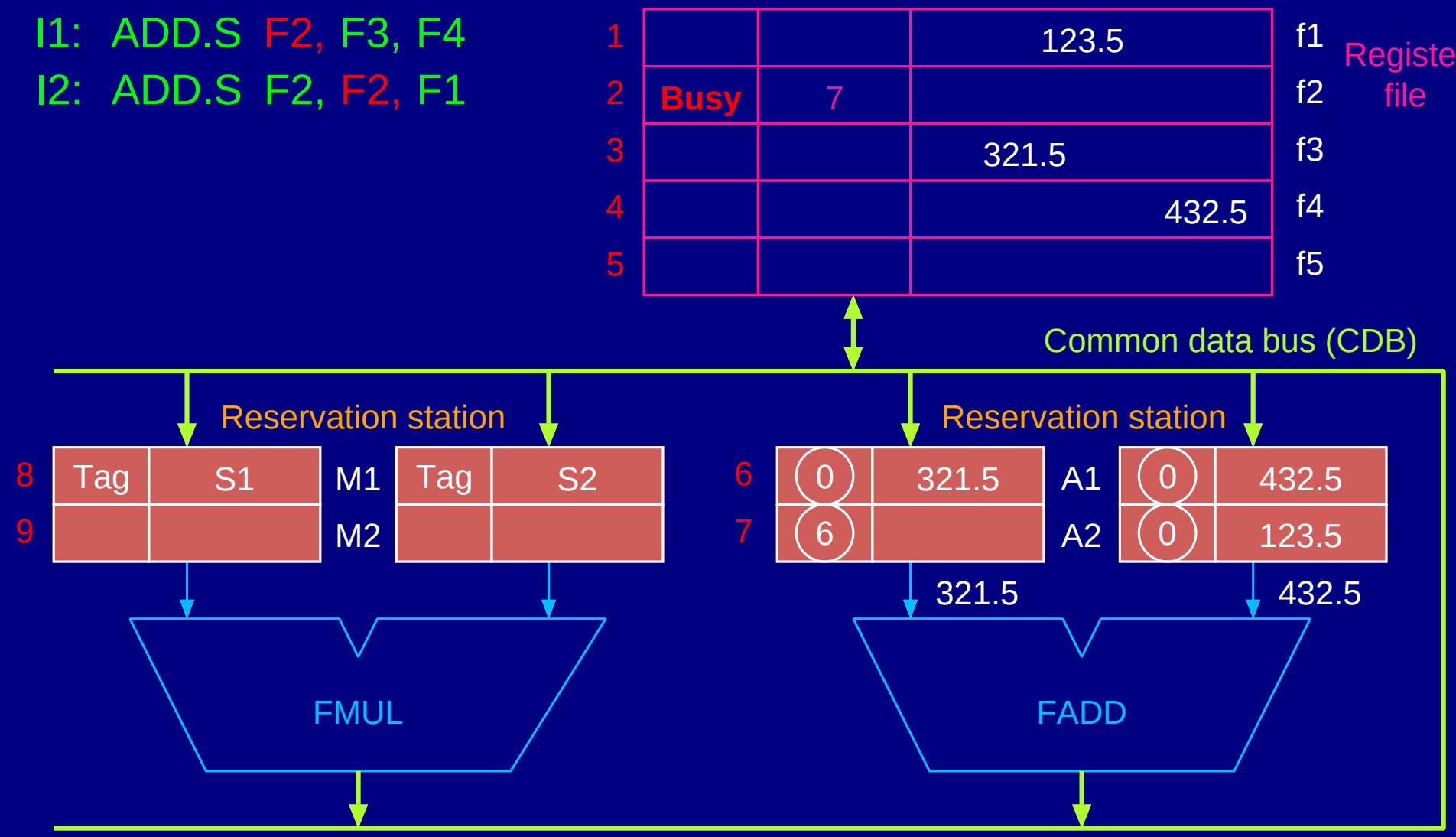
Clock Cycle 3 — Executing I1 (2)

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



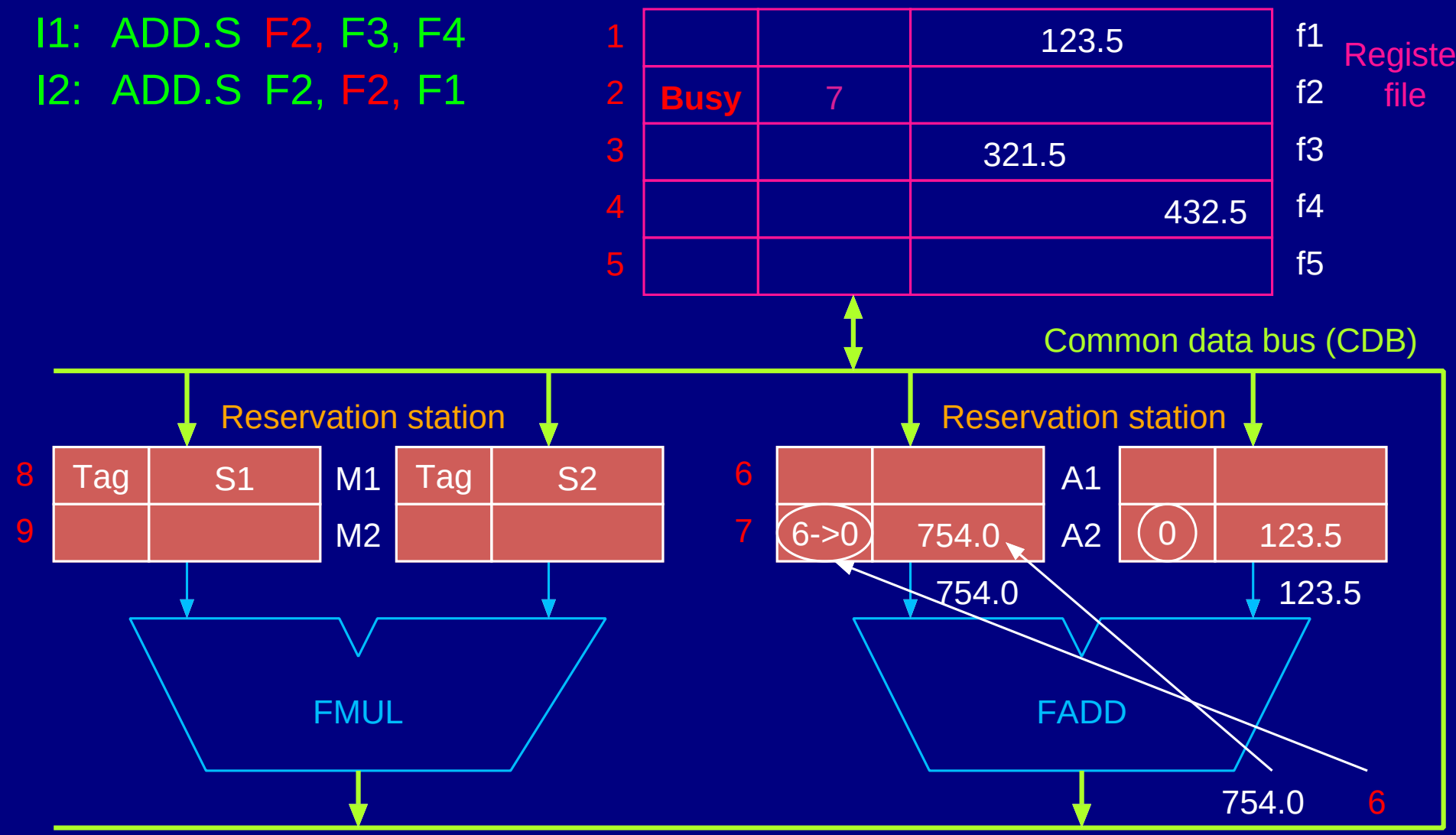
Clock Cycle 4 — Executing I1 (3)

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



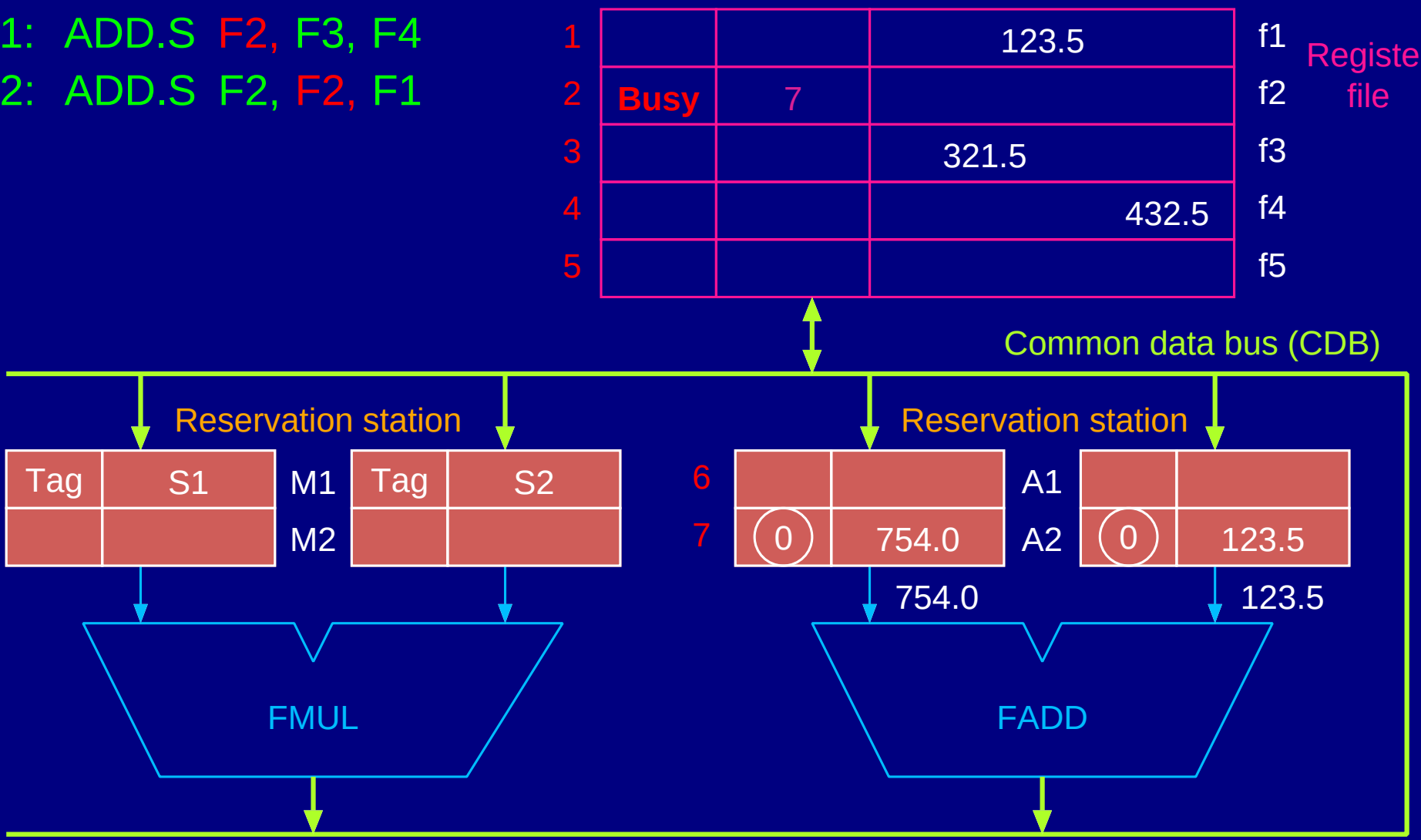
Clock Cycle 5 — Writing I1, Executing I2 (1)

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



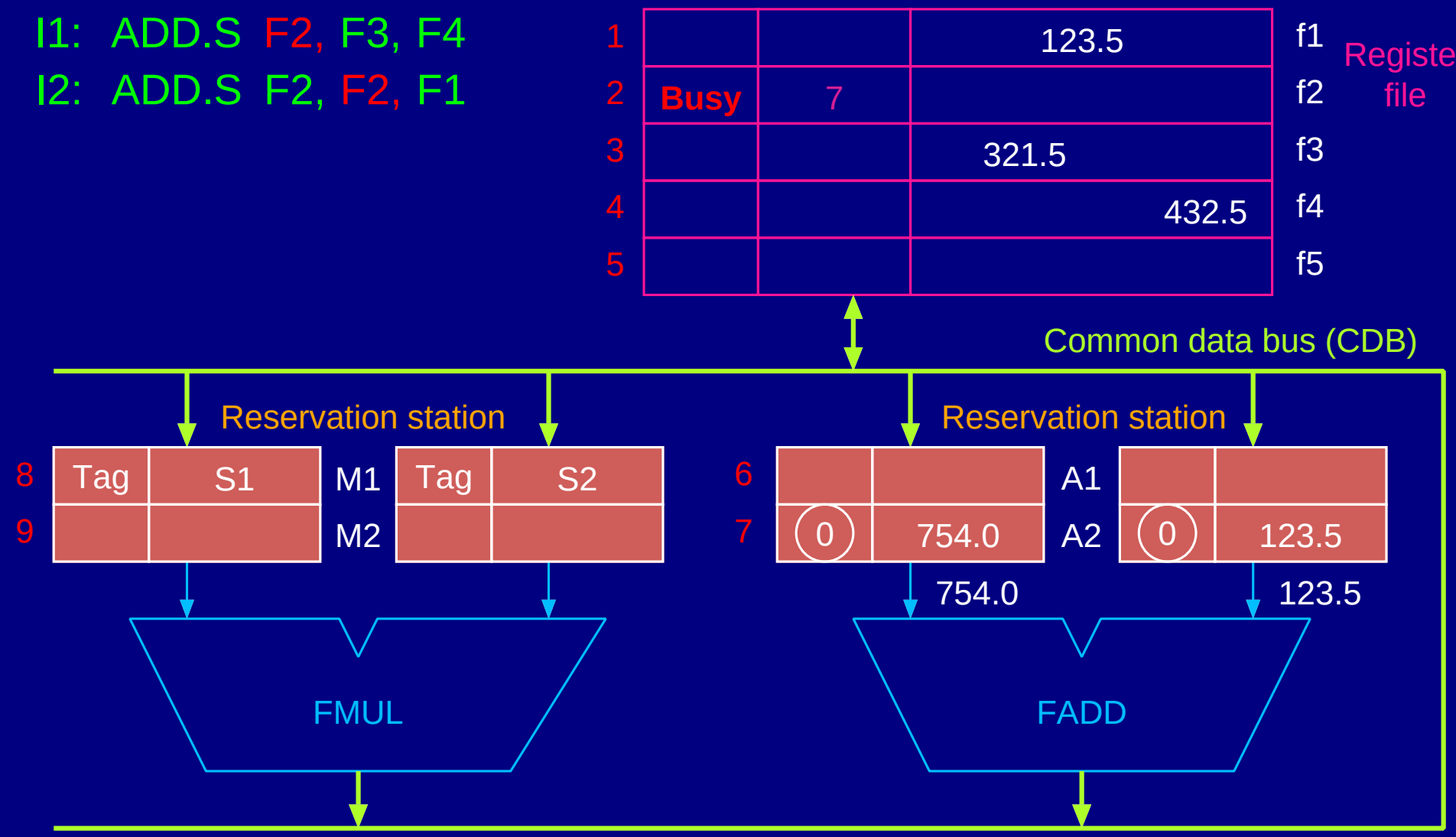
Clock Cycle 6 — Executing I2 (2)

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



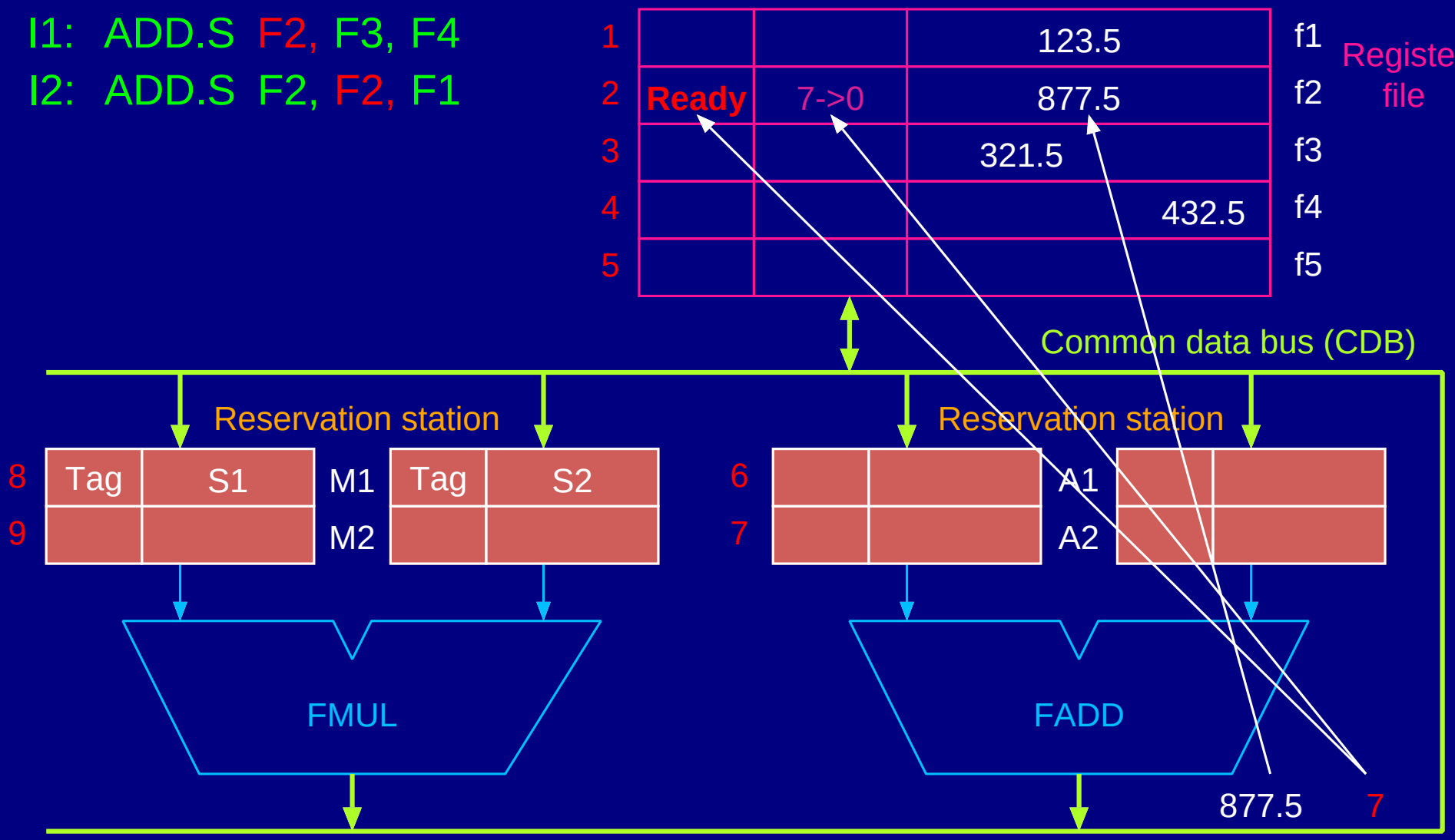
Clock Cycle 7 — Executing I2 (3)

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



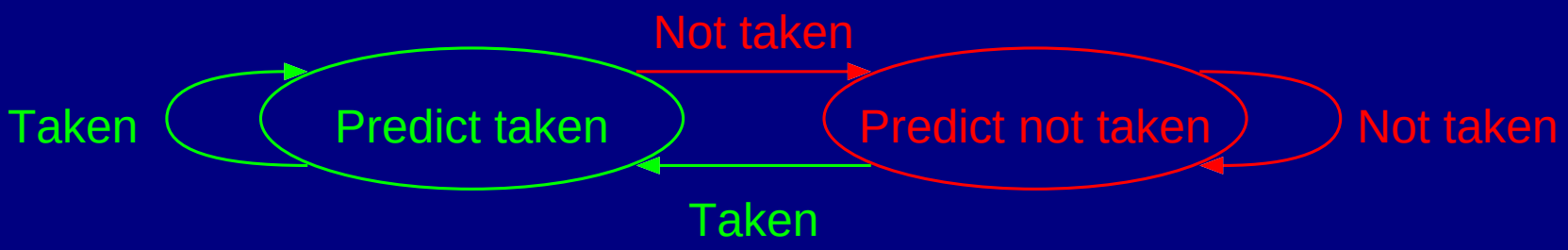
Clock Cycle 8 — Writing I2

I1: ADD.S F2, F3, F4
I2: ADD.S F2, F2, F1



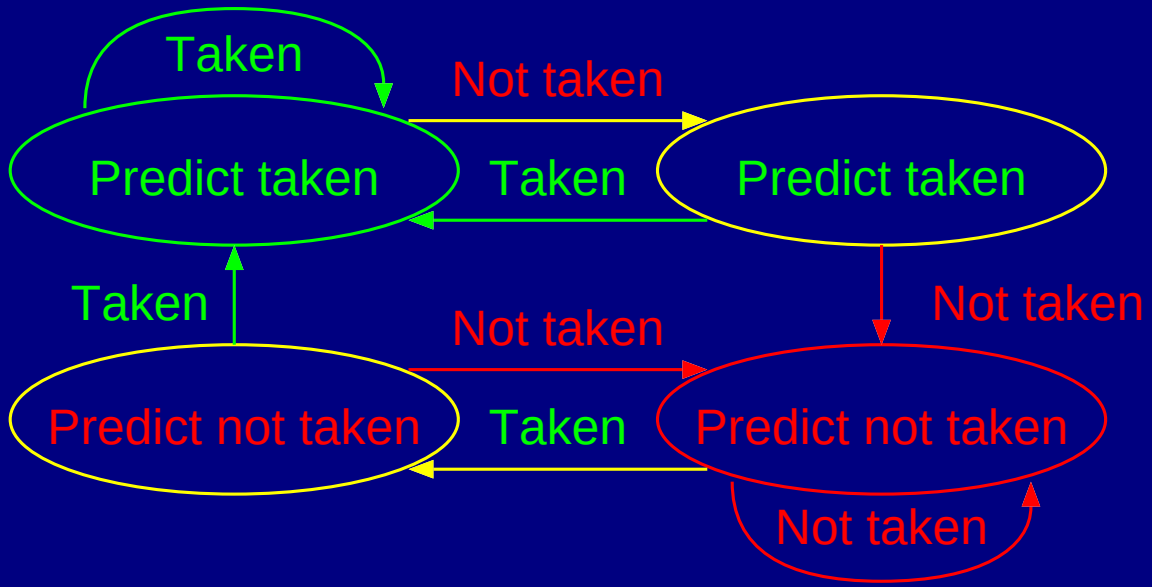
Dynamic Hardware Branch Prediction

- Branch Prediction
  - Static Software Branch Prediction
  - Dynamic Hardware Branch Prediction
    - A branch-prediction buffer is a small memory indexed by the low portion of the address of the branch instruction.
    - The memory contains a bit that says whether the branch was recently taken or not.
- One-bit Prediction Scheme

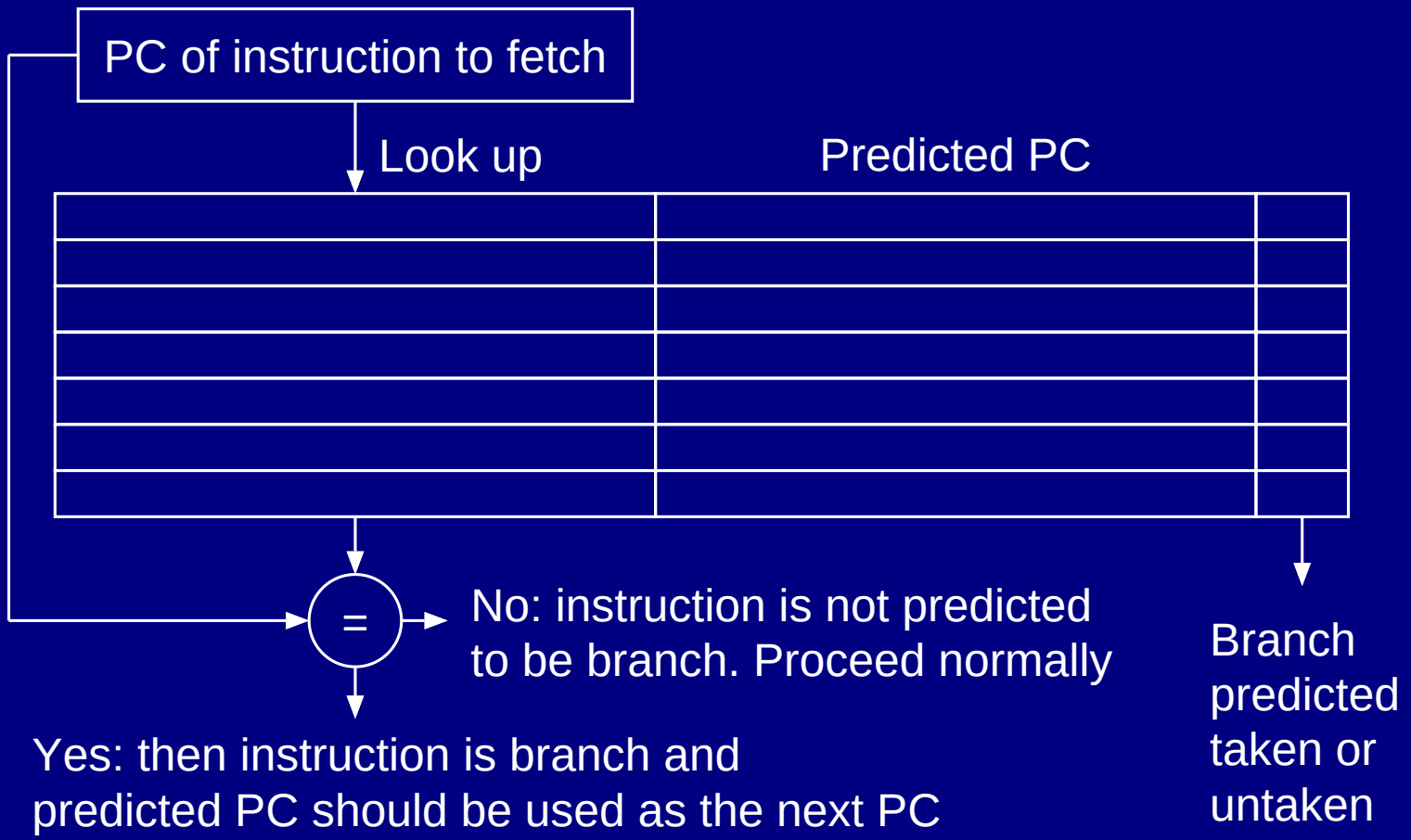


Two-bit Prediction Scheme

- In a two-bit scheme, a prediction must miss twice before it is changed:



Branch Target Buffer (BTB)



Performance Improvement of Using BTB

- P_b: the frequency an instruction is a branch;
- P_t: the frequency branch is taken;
- P_h: the hit ratio in the buffer;
- P_p: the prediction accuracy;
- C: the penalty cycles when misprediction or when the buffer is miss and branch is taken;
- T: The average clock cycles per instruction when BTB is not used. Then

$$T = P_b(P_t\,(1 + C) + (1 - p_t) \times 1) + (1 - P_b) \times 1$$

Performance Improvement of Using BTB

- T_{BTB}: The average clock cycles per instruction when BTB is used. Then

$$\begin{aligned} T_{BTB} &= P_b\, B + (1 - P_b) \times 1 \\ B &= P_h\, H + (1 - p_h)\, M \\ H &= P_p \times 1 + (1 - P_p)(1 + C) \\ M &= P_t\, (1 + C) + (1 - P_t) \times 1 \end{aligned}$$

- Example:
  $P_b = 0.2,\ P_t = 0.6,\ P_h = P_P = 0.9,\ C = 2$
- $T = 1.24$, $T_{BTB} = 1.06$
- Speedup $= 1.24/1.06 = 1.17$ times

Software-Based Speculation

- Consider the code fragment

  if (A==0) then A=B; else A=A+4;

- If A is at 0(R3) and B is at 0(R2):

```
start:  L.W     R1, 0(R3)    ; load A
if:     BNEZ    R1, else     ; test A
then:   L.W     R1, 0(R2)    ; then clause
        J       join         ; skip else
else:   DADDI   R1, R1, #4   ; else clause
join:   S.W     0(R3), R1    ; store A
```

Software-Based Speculation

- Assume the then clause is *almost always* executed and R14 is unused and available. Then

```
start:  L.W     R1, 0(R3)     ; load A
then:   L.W     R14, 0(R2)    ; then clause
if:     BEQZ    R1, join      ; test A
else:   DADDI   R14, R1, #4   ; else clause
join:   S.W     0(R3), R14    ; store A
```

- This type of code scheduling is called *speculation*, since the compiler is basically betting on the branch outcome; in this case that the branch is usually taken.

Hardware-Based Speculation

- Hardware-based speculation combines three key ideas:
  1. *dynamic branch prediction* to choose which instructions to execute;
  2. *speculation* to allow the execution of instructions before the control dependences are solved; and
  3. *dynamic scheduling* to deal with the scheduling of different combinations of basic blocks.
- This method of executing programs is essentially a *data-flow execution*: operations execute as soon as their operands are available.

Hardware-Based Speculation

- The key idea behind implementing speculation is to allow instructions to execute *out-of-order* but to force them to commit *in-order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.
- Adding this commit stage, we need an set of hardware buffer to hold the results of instructions that have finished execution but have not committed.
- This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

Multiple-Issue Processors

- The techniques we discussed up to now can be used to eliminate data and control stalls and achieve an ideal CPI of 1.
- To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.
- The goal of the *multiple-issue processors* is to allow multiple instructions to issue in a clock cycle.
- Multiple-issue processors come in two flavors: *superscalar* processors and *VLIW* (very long instruction word) processors.

Superscalar Processors

Clock	1	2	3	4	5	6	7	8	9
Inst i	IF	ID	EX	MEM	WB				
Inst i+1	IF	ID	EX	MEM	WB				
Inst i+2		IF	ID	EX	MEM	WB			
Inst i+3		IF	ID	EX	MEM	WB			
Inst i+4			IF	ID	EX	MEM	WB		
Inst i+5			IF	ID	EX	MEM	WB		
Inst i+6				IF	ID	EX	MEM	WB	
Inst i+7				IF	ID	EX	MEM	WB	
Inst i+8					IF	ID	EX	MEM	WB
Inst i+9					IF	ID	EX	MEM	WB

Superscalar Pipeline Example

	Integer instruction		FP instruction		Clock cycle
Loop:	L.D	F0, 0(R1)			1
	L.D	F6, 8(R1)			2
	L.D	F10, 16(R1)	ADD.D	F4, F0, F2	3
	L.D	F14, 24(R1)	ADD.D	F8, F6, F2	4
	L.D	F18, 32(R1)	ADD.D	F12, F10, F2	5
	DADDI	R2, R2, #-5	ADD.D	F16, F14, F2	6
	S.D	0(R1), F4	ADD.D	F20, F18, F2	7
	S.D	8(R1), F8			8
	S.D	16(R1), F12			9
	S.D	24(R1), F16			10
	S.D	32(R1), F20			11
	BNEZ	R2, Loop			12
	DADDI	R1, R1, #40			13

Execution time is reduced to 13/5, or 2.6 clock cycles.

VLIW Processors

- VLIW processors issue a fixed number of instructions formed either as one large instruction or as a fixed instruction packet to multiple independent functional units.
- VLIW processors are inherently statically scheduled by the compiler. The compiler has complete responsibility for creating a package of instructions that can be simultaneously issued.
- The parallelism is uncovered by unrolling loops and scheduling code across basic blocks using a global scheduling technique. One of these technique is called *trace scheduling*, developed specifically for VLIWs.

VLIW Example

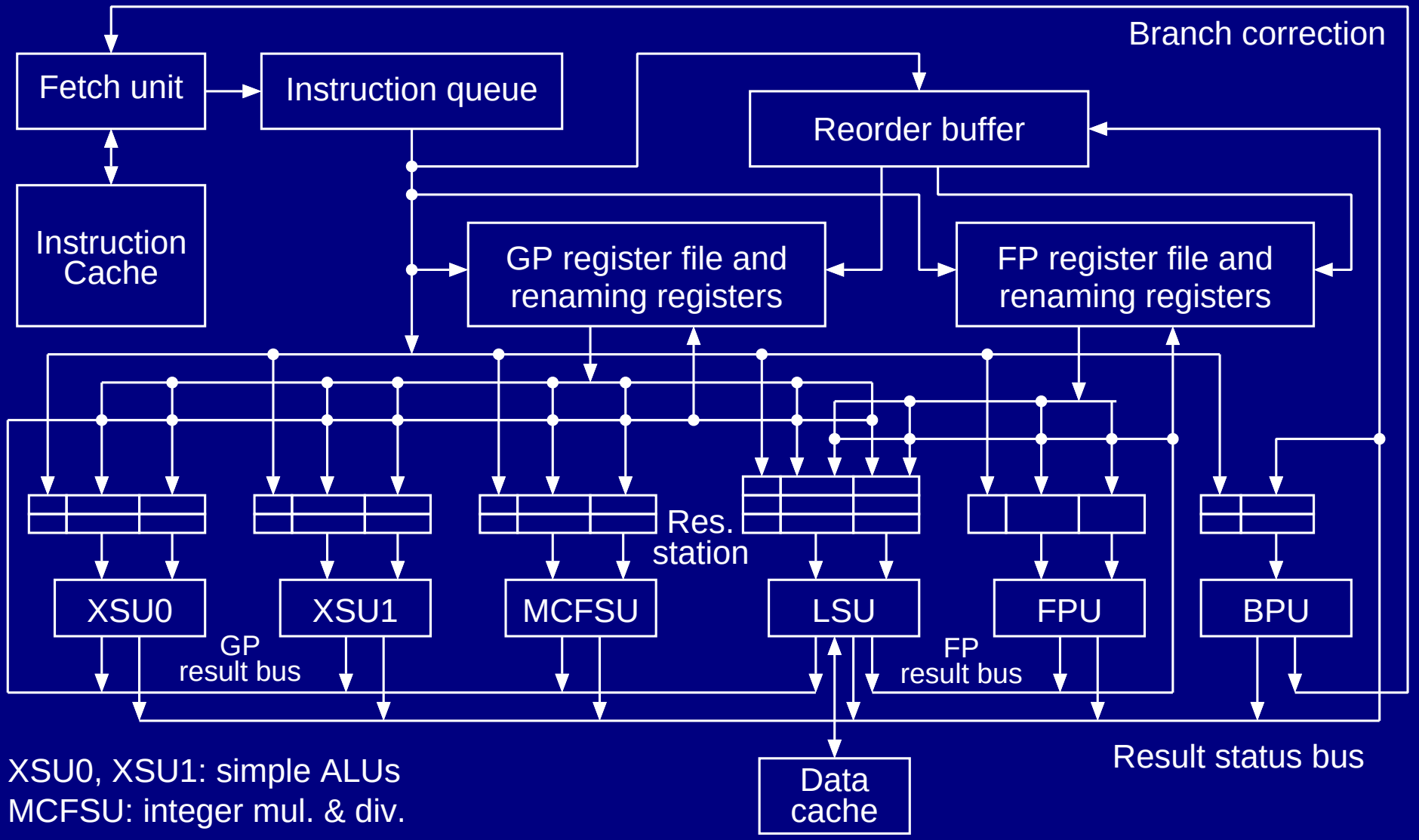
Memory reference 1	Memory reference 1	FP operation1	FP operation2	Integer operation
L.D F0, 0(R1)	L.D F6, 8(R1)			
L.D F10, 16(R1)	L.D F14, 24(R1)	ADD.D F4, F0, F2	ADD.D F8, F6, F2	
L.D F18, 32(R1)	L.D F22, 40(R1)	ADD.D F12, F10, F2	ADD.D F16, F14, F2	
L.D F26, 48(R1)		ADD.D F20, F18, F2	ADD.D F24, F22, F2	
		ADD.D F28, F26, F2		DADDI R2, R2, #-7
S.D 0(R1), F4	S.D 8(R1), F8			
S.D 16(R1), F12	S.D 24(R1), F16			
S.D 32(R1), F20	S.D 40(R1), F24			BNEZ R2, Loop
S.D 48(R1), F28				DADDI R1, R1, #56

Execution time is reduced to 9/7, or 1.3 clock cycles.

The PowerPC 620

- The PowerPC 620 is an implementation of the 64-bit version of the PowerPC architecture.
- This implementation embodies many of the ideas we discussed, including out-of-order execution and speculation.
- The 620 can fetch, issue, and complete up to four instructions per clock cycle.
- The reorder buffer contains only the information needed to complete the instruction when it commits.
- There are 8 integer rename registers and 12 floating point rename registers.

The PowerPC 620



XSU0, XSU1: simple ALUs
MCFSU: integer mul. & div.

The IA-32 Intel Architecture

- The IA-32 Intel Architecture, from the Intel 8086 processor to the latest version implemented in the Pentium 4 and Intel Xeon processors, has been at the forefront of the computer revolution and is today the preferred computer architecture.
- The two major factors that drive the popularity of IA-32 architecture are:
  - software compatibility
  - and the fact that each generation of the IA-32 processors delivers significantly higher performance.

Intel IA-32 Processors

Intel processor	Date intro-duced	Micro-archite-cture	Clock	Tran-sistors per die	Register size	Ext. data bus	Exern. addr. space	On-die caches
8086	1978		8 MHz	29 K	GP: 16	16 bits	1 MB	None
286	1982		12.5 MHz	134 K	GP: 16	16 bits	16 MB	
386 DX	1985		20 MHz	275 K	GP: 32	32 bits	4 GB	
486 DX	1989		25 MHz	1.2 M	GP: 32	32 bits	4 GB	L1: 8KB
Pentium	1993		60 MHz	3.1 M	GP: 32 FPU: 80	64 bits	4 GB	L1: 16KB
Pentium Pro	1995	P6	200 MHz	5.5 M	GP: 32 FPU: 80	64 bits	64 GB	L1: 8KB L2: 256/512KB
Pentium II	1997	P6	266 MHz	7.0 M	GP: 32 FPU: 80 MMX: 64	64 bits	64 GB	L1: 8KB L2: 256/512KB
Pentium III	1999	P6	500 MHz	8.2 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	64 bits	64 GB	L1: 8KB L2: 256KB

Intel IA-32 Processors

Intel processor	Date intro-duced	Micro-archite-cture	Clock	Tran-sistors per die	Register size	Sys. bus band-width	Exern. addr. space	On-die caches
Pentium III and Pentium III Xeon	1999	P6	700 MHz	28 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	Up to 1.06 GB/s	64 GB	L1: 3KB L2: 256KB
Pentium 4	2000	NetBurst	1.5 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K μop trace cache L1: 8KB L2: 256KB
Xeon	2001	NetBurst	1.7 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K μop trace cache L1: 8KB L2: 256KB

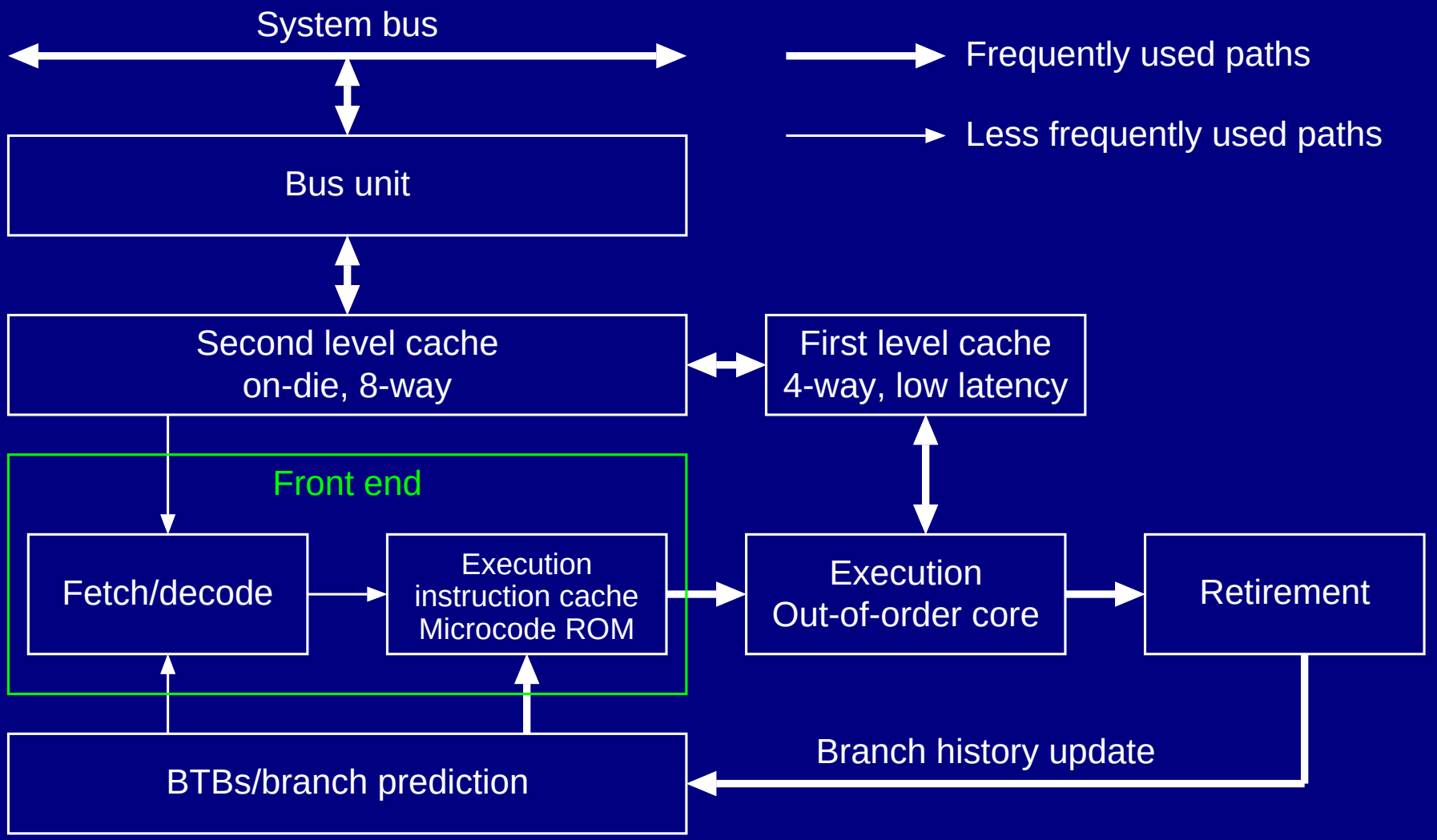
Intel IA-32 Processors

Intel processor	Date intro-duced	Micro-archite-cture	Clock	Tran-sistors per die	Register size	Sys. bus band-width	Exern. addr. space	On-die caches
Xeon	2002	NetBurst Hyper-Threading	2.2 GHz	55 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K μop trace cache L1: 8KB L2: 512KB
Xeon MP	2002	NetBurst Hyper-Threading	1.6 GHz	108 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K μop trace cache L1: 8KB L2: 512KB
Pentium 4 supporting Hyper-Threading	2002	NetBurst Hyper-Threading	3.06GHz	55M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K μop trace cache L1: 8KB L2: 512KB

The Intel P6 Microarchitecture

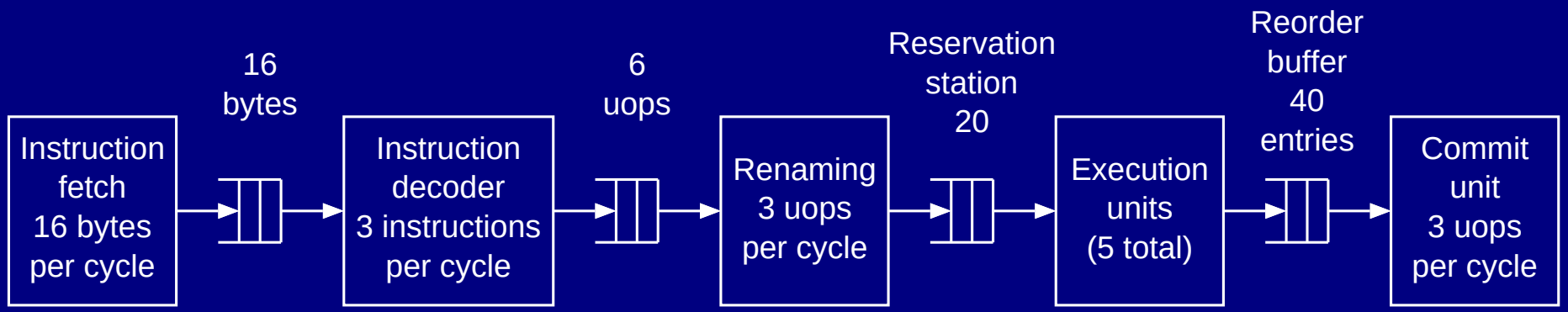
- The Intel P6 microarchitecture forms the basis for the Pentium Pro, Pentium II, and Pentium III.
- In addition to some specialized instruction set extensions (MMX, and SSE), these three processors differ in clock rate, cache architecture, and memory interface, as summarized in the tables.
- The P6 microarchitecture is a dynamically scheduled processor that translates each IA-32 instruction into a series of micro-operations (uops).
- The uops are executed by an out-of-order speculative pipeline using register renaming and a reorder buffer.

The P6 Processor Microarchitecture



The P6 High Level Pipeline

- Up to three uops per cycle can be renamed and dispatched to the reservation stations.
- Instruction commit can also complete up to three uops per cycle.
- A high-level picture of the P6 pipeline:



Pentium 4 Processor